

UNIVERSITY OF
ILLINOIS LIBRARY
AT URBANA-CHAMPAIGN
ENGINEERING

NOTICE: Return or renew all Library Materials! The *Minimum Fee* for each Lost Book is \$50.00.

JUL 06 1988

The person charging this material is responsible for its return to the library from which it was withdrawn on or before the **Latest Date** stamped below.

Theft, mutilation, and underlining of books are reasons for disciplinary action and may result in dismissal from the University.

To renew call Telephone Center 833-6400

UNIVERSITY OF ILLINOIS LIBRARY AT URBANA-CHAMPAIGN

L161—O-1096

10.84 ENGINEERING LIBRARY
163e UNIVERSITY OF ILLINOIS
0.236 URBANA, ILLINOIS

Engine

CONFERENCE ROOM

Center for Advanced Computation

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
URBANA, ILLINOIS 61801

CAC Document Number 236
CCTC-WAD Document Number 7616

Research in
Network Data Management and
Resource Sharing

**INTELLIGENT TERMINAL
PROGRAMMER'S MANUAL**

Volume One of Two Volumes

October 31, 1977

The Library of the

MAY 23 1978

University of Illinois

The person charging this material is responsible for its return to the library from which it was withdrawn on or before the **Latest Date** stamped below.

Theft, mutilation, and underlining of books are reasons for disciplinary action and may result in dismissal from the University.

UNIVERSITY OF ILLINOIS LIBRARY AT URBANA-CHAMPAIGN

ENGINEERING

JUN 6 1978

MAR 7 1982

MAR 3 RECD

CAC Document Number 236
CCTC-WAD Document Number 7516

Intelligent Terminal
Programmer's Manual

Volume One of Two Volumes

Deborah S. Brown
Daniel J. Kopetzky
John R. Mullen
David A. Willcox

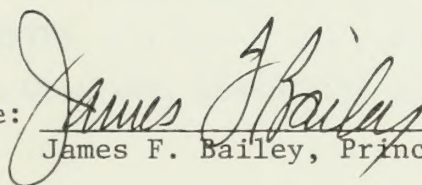
Prepared for the
Command and Control Technical Center
WWMCCS ADP Directorate
Defense Communication Agency
Washington, D.C.

under contract
DCA100-76-C-0088


Center for Advanced Computation
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

October 31, 1977

Approved for release:



James F. Bailey, Principal Investigator



Digitized by the Internet Archive
in 2012 with funding from
University of Illinois Urbana-Champaign

<http://archive.org/details/intelligenttermi00brow>

Table of Contents

	Page
INTRODUCTION	1
Structure of Manual	1
Assumptions	2
Reader	2
Support Facilities	2
Hardware	3
Software	3
Documentation	4
Un-Goals	4
 PART I: IT HARDWARE AND SOFTWARE	
OVERVIEW	7
HARDWARE CONFIGURATION	9
LSI-11 IT	9
Processor	9
Storage	9
Display	9
Touch Panel	11
Keyboard/Control Console	11
Communication Device	11
Level 6 IT	12
Processor	12
Storage	12
Remote Display Head	12
Communication Devices	12
SYSTEM OVERVIEW	15
Operating System Kernel	16
Support Software	18
I/O System	18
Application Support	19
OPERATING SYSTEM	21
Process Creation, Scheduling and Removal	21

System Initialization	24
LSI-11 System Initialization	24
Level 6 System Initialization	26
Process Synchronization	27
Queues	28
Scheduler	31
Memory Management	32
Interrupts and Traps	32
LSI-11 Mechanism	32
Level 6 Mechanism	33
Errors	34
C Program Support	34
I/O SYSTEM AND DEVICE HANDLERS	35
I/O System	35
Device Handlers	38
Purpose	38
Structure	39
Interrupt Routines	39
Naming Conventions	41
Device Names	42
File Names	42
APPLICATION SUPPORT SOFTWARE	45
Plasma Panel	45
Graphics	45
Printing	46
Level 6 Interface	49
Touch Targets	50
General Structure	51
Manipulating Touch Targets	52
Notes	53
String Manipulation	54
Searching Strings	54
String Parsing	55
String Formatting and Conversion	55

Data Display	55
Table	57
Bar Graph	57
Map	58

PART II: BUILDING AND EXECUTING IT SYSTEMS

OVERVIEW	63
COOKBOOK	65
Introduction	65
LSI-11 IT	65
Assumptions	65
Building an Executable IT System	66
Loading the System onto the IT	69
Level 6 IT	73
Assumptions	73
Building an Executable IT System	74
Loading the System onto the IT	77
BUILDING AN IT SYSTEM	79
Introduction	79
Building an LSI-11 IT System	79
Naming Conventions	80
Creating Source Files	80
Creating Loadable Object Files	81
Building Libraries	81
Loading the Executable System	84
Some Random Notes on Building Systems	85
Building a Level 6 IT System	86
Naming Conventions	86
Creating Source Files	88
Creating Assembly Language Files	88
Transferring Files to the Level 6	88
Assembling the Files	90
Linking the Executable System	91
Some Notes on Building Systems	93

LOADING A COMPLETED SYSTEM ONTO THE IT	95
Introduction	95
Loading the LSI-11 IT	95
Using Communication Line	95
Using Floppy Disk Bootloader	97
Using SOTS	98
Comparision of Loading Methods	98
Loading the Level 6 IT	99
Using BES Command Processor	100
Using <u>BTGEN</u> on Disk	100
Comparision of Loading Methods	100
DEBUGGING IT SYSTEMS	103
Getting Started	103
Accessing Memory	103
UNIX nm	103
Level 6 MAP	104
Useful Variables	104
ME	104
READY_Q	105
FREE_PTR	105
PROCTAB	105
DEV_TAB	105
KBD_Q, PP_Q, TP_Q, VIP_Q, DSK_Q	105
Registers	106
IT Process Stacks	106
C Conventions	106
Process Stacks	115
IT Post-Mortem	116
 PART III: MAINTAINING IT SYSTEM SOFTWARE	
OVERVIEW	133
MODIFYING THE I/O SYSTEM	135
Adding I/O Functions	135
Create the Handling Routine	135
Modify Device Handlers	138
Update System Library	139

Adding a Device	139
Interrupt Handlers	141
I/O System Modifications	147
Device Handler	147
System Modification	149
DISK ACCESSING SOFTWARE	151
Disk Format	151
Physical Structure of Disk	151
Logical Structure of Disk	152
Reserved Blocks	152
File Structure	153
Directory Structure	153
Code Structure	156
I/O System	158
Disk Driver	158
Primary File Level Routines	158
File Level Support Routines	161
Block Level Routines	161
Sector Level Routines	162
ACCESSING REMOTE DISPLAY HEAD	165
Driving the Plasma Panel	165
Touch Panel/Keyboard Input	167
BIBLIOGRAPHY	169

APPENDIX A: IT Procedures by Functional Grouping

APPENDIX B: Description of IT Procedures

APPENDIX C: Description of IT Data Structures

APPENDIX D: Character Set Description

APPENDIX E: Interfaces to Plasma and Touch Panels

Table of Figures

Figure	Page
1 UNIX directory structure of IT files	3
2 LSI-11 IT	10
3 Level 6 IT	13
4 Hierarchial structure of IT system code	15
5 How user process accesses device	18
6 Stack of process initialized by creep on LSI-11 IT	23
7 Stack of process initialized by creep on Level 6 IT	23
8 Structure of PROCTAB	25
9 Structure of a semaphore	27
10 Queue structure	29
11 Pool of free queue elements	30
12 Structure of queue of ready processes	31
13 Structure of DEV_TAB entry	37
14 File naming conventions	44
15 Ld using a library	83
16 Subroutine environment linkage - LSI-11	108
17 Subroutine environment linkage - Level 6	110
18 Snapshot of LSI-11 process stack with local variables	111
19 Snapshot of Level 6 process stack with local variables	112
20 Subroutine linkage - LSI-11	113
21 Subroutine linkage - Level 6	114
22 Process stack - LSI-11 IT	117
23 Process stack - Level 6 IT	118
24 Flow of information and control between processes, interrupt handler, and device	140

Table of Figures
(continued)

Figure	Page
25 Structure of a file	154
26 Structure of a directory entry	155
27 Example of disk directory	157

List of Tables

Table	Page
1 System Namelist	120
2 Level 6 plasma panel task codes and parameters	166

INTRODUCTION

This document is designed to be a reference manual for programmers writing code to be run on an Intelligent Terminal (IT) as implemented by the Center for Advanced Computation (CAC) of the University of Illinois. It serves as an introduction to the IT and its existing software. The manual also outlines the procedure for utilizing and modifying existing IT code.

The IT has been a research project at the Center for over two years. It was designed to be an experiment in the area of using intelligent terminals as active user agents in interfacing to existing data management systems. To this end, the IT, consisting of a single-user microprocessor based terminal, was built. The terminal is supported by software including a multi-processing operating system and several support routines. This document attempts to describe that software sufficiently that other programmers can utilize the existing code in their work with the IT.

Structure of Manual

The body of this manual is divided into three major parts. The first part is an overall view of the IT and its existing software. Part I is designed as the programmer's introduction to the IT. The second part gives detailed explanations of how to build an IT system and how to load it into the terminal's memory. It includes a step by step procedure as well as a discussion of the why of each step. The third part of the manual contains more detailed discussions of certain portions of the IT software. The sections in this part are designed for programmers who need to modify the IT system code or to understand it on a detailed level.

Within the three major parts of the manual, many discussions are presented twice. This is necessary since there are two Intelligent Terminals described here. The two ITs which have been built are based on different minicomputers, and are physically very different. The software functions described in this manual are provided for both terminals. However, the implementation details of individual functions may differ widely between the two systems. In these cases, the methods for both terminals are presented.

Many of the sections of this manual make various assumptions regarding the knowledge of the reader and the availability of support. These assumptions are stated at the beginning of each part or section. The major assumptions made throughout the manual are discussed below. Following these is a disclaimer of what this manual is not.

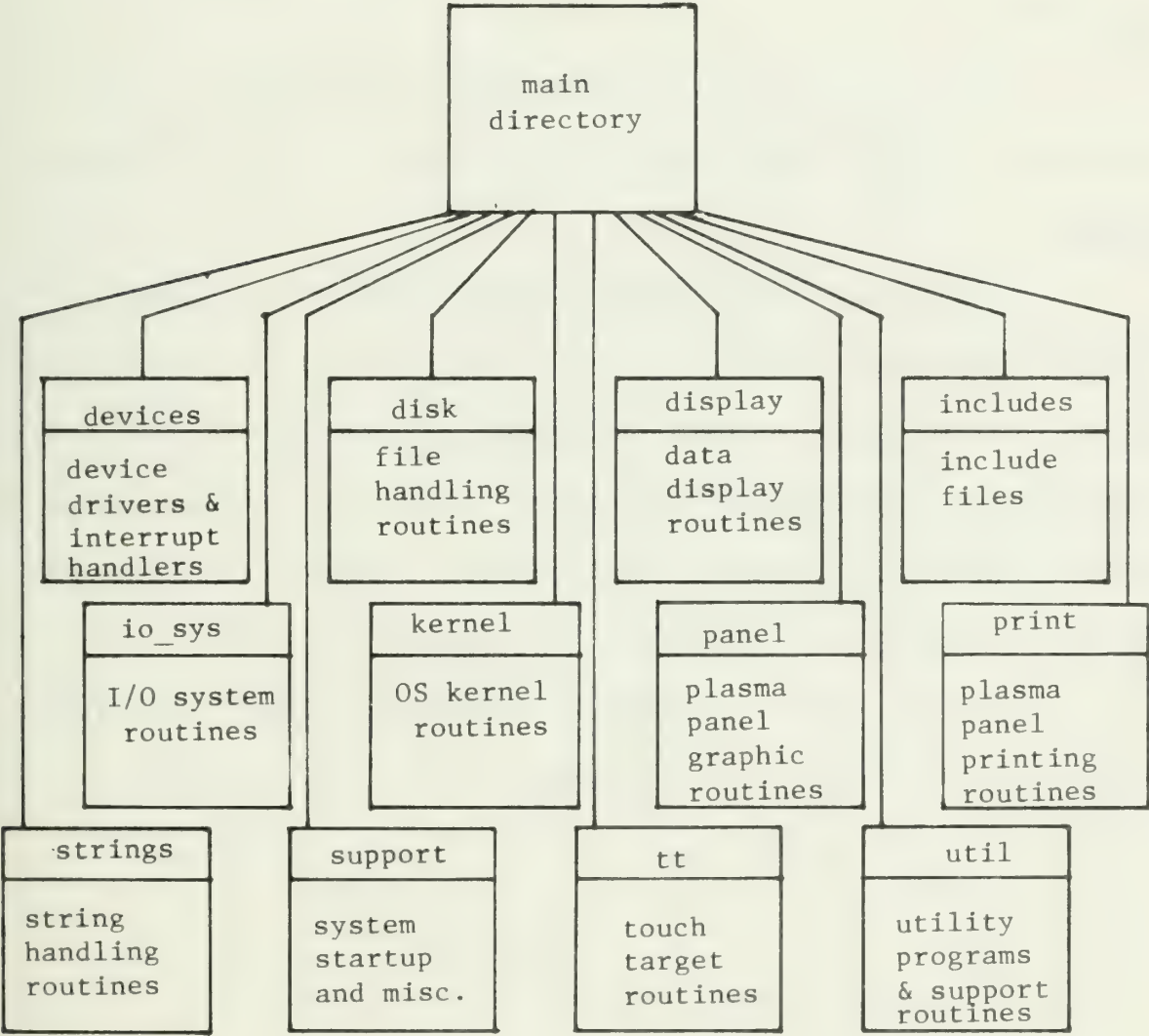
Assumptions

Reader. The reader of this document is assumed to be a professional computer programmer interested in utilizing existing IT software to support a system on a CAC Intelligent Terminal. Further the reader is assumed to be somewhat familiar with the UNIX system, the C language, and the operation and hardware of the PDP LSI-11 and Honeywell Level 6.

Support Facilities. It is assumed that the reader will have access to a PDP-11 system running the UNIX operating system developed at Bell Laboratories. In order to utilize existing IT software, which is written in C, this system must support the C compilers for the LSI-11 and the Level 6, as well as and other functions necessary to compile and load C language programs. Additionally, the UNIX system must have undergone the modifications which allow it to read and write floppy disks in the IT format.

Hardware. The programmer will need access to a standard CAC LSI-11 or Level 6 based Intelligent Terminal. LSI-11 terminals must be capable of being connected to the machine running UNIX, either by a hardwired connection or by a modem and phone line.

Software. The programmer is assumed to have access to the source code of all procedures and data structures mentioned in this document. For ease of reference, this code is assumed to be organized on UNIX into a standard set of subdirectories off of one main directory. This directory structure and the contents of each directory is diagrammed in Figure 1.



UNIX directory structure of IT files

Figure 1

Documentation. In addition to this manual, the reader is assumed to have access to:

1. the UNIX Programmer's Manual [11],
2. the C Reference Manual [10],
3. the LSI-11 PDP-11/03 Processor Handbook [7],
4. Honeywell Level 6 Minicomputer Handbook [6], and
5. Honeywell Level 6 Operator's Guide [8].

Further it is assumed that the programmer will spend time with listings of the current IT software, in addition to this manual, to gain a detailed level of understanding before trying to implement any changes to the system.

Within this manual, the names of IT procedures, IT data structures, and UNIX directories are underlined to reduce ambiguity.

Un-goals

This manual is not designed to be a guide to the types of systems to be run on the IT. It merely describes the existing software and how to get the pieces into one usable system after they are built. The problem of designing and building the application package to go on the IT is left as an exercise for the reader. CAC's experiences with application systems are described in portions of [2] and [3].

Finally this paper makes no attempt to justify the design decisions that went into creating the IT.

PART I:

IT HARDWARE AND SOFTWARE

OVERVIEW

Part I of this manual is concerned with existing ITs. It includes:

1. descriptions of the hardware for the LSI-11 IT and the Level 6 IT,
2. a general description of the IT software, and
3. more detailed descriptions of the IT operating system, the I/O system and application support routines.

Hardware and software for both the LSI-11 and the Level 6 based ITs is covered. Discrepancies between the two versions are noted and described.

The sections in Part I are designed to be a tutorial describing the types of tools available to an IT programmer and constraints on the use of these tools. Part I should be read, or at least scanned, sequentially before the rest of the manual and before attempting to use or modify the IT system.

HARDWARE CONFIGURATION

This section describes the hardware comprising an IT. The hardware is significantly different for an LSI-11 IT than for a Level 6 IT. Accordingly each terminal is described independently. The LSI-11 IT is described first and then the Level 6 IT.

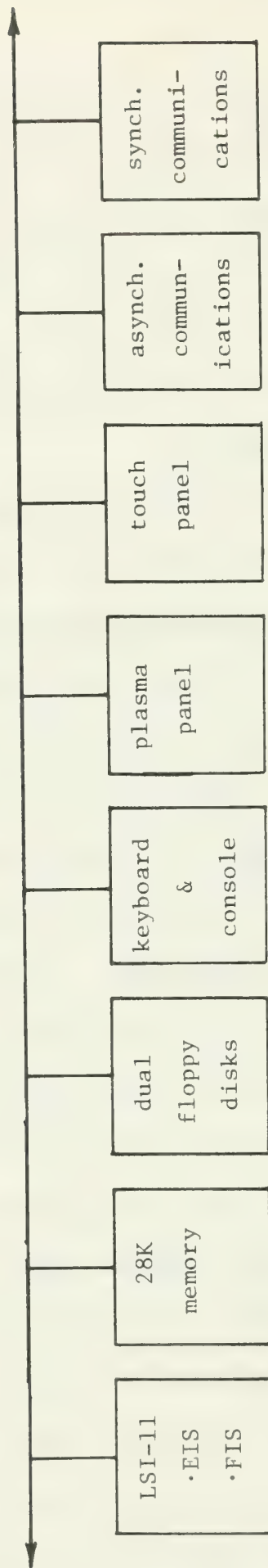
LSI-11 IT

The layout of this terminal is illustrated in Figure 2. Its components are described below.

Processor. This Intelligent Terminal system is based on the Digital Equipment Corporation (DEC) LSI-11 processor. The LSI-11 is a new member of DEC's PDP-11 family of computer systems. It is compatible with previous members of that family; its instruction set is a duplicate of the earlier PDP-11/40. The outstanding features of the LSI-11 are its size and speed, as well as its extended arithmetic and floating point instruction set capabilities. The processor is contained on a single printed circuit card measuring 10" by 11". The processor card includes 4K words of random access memory (RAM). However, this 4K section of memory is not utilized due to the heavy demands it places on the system. Further details of the processor can be found in [7].

Storage. The CAC LSI 11 system includes a total of 28K words of RAM memory (exclusive of the 4K that is integral with the processor). An Advanced Electronics Design (AED) 3100LP floppy disk system providing 500K bytes of removable storage is also included.

Display. The display device used on the IT is an Owens-Illinois D-142 plasma panel. The plasma panel is a flat-screen display composed of a 512 x 512 matrix of dots. Each dot may be turned on or off independently from any other dot. The plasma panel is interfaced to



LSI-11 IT

Figure 2

the LSI-11 by means of a custom design controller developed at CAC. It was necessary to develop this special interface due to the lack of a suitable commercial product.

The D-142 plasma panel is an unsophisticated device in that it provides no intrinsic character generation or graphic display capability; these functions must be supplied by software. Characters of an alphabet are stored in software as a series of bit masks. Since the shape of a character is determined by its associated bit patterns, the text display software can be used to display arbitrary figures formed as a series of characters.

Touch Panel. The primary input device for the IT is a 32 x 32 touch panel from Carroll Manufacturing Corporation. This device consists of a grid of 32 pairs of intersecting light beams, and is located on the face of the plasma panel display. When the user touches a spot on the display, the touch panel determines the location of his finger and transmits that location to the software. The software can then perform various functions depending on the location the user touched. Proper coordination between displays and touches allow the user to utilize the Intelligent Terminal without the necessity of typing commands to it.

Keyboard/Control Console. An auxilliary input capability is also provided by a serial-output, microprocessor controlled keyboard. This keyboard has been modified so that it can also serve as the system operator console.

Communication Device. Asynchronous communication between the IT and a host computer is provided via a standard RS232C interface. This interface can also be used to connect to a standard modem.

In addition, there is a synchronous interface suitable for connecting the IT to a 6000 VIP port. The interface is user-settable to either RS232 or MIL-STD-188C signal levels.

Level 6 IT

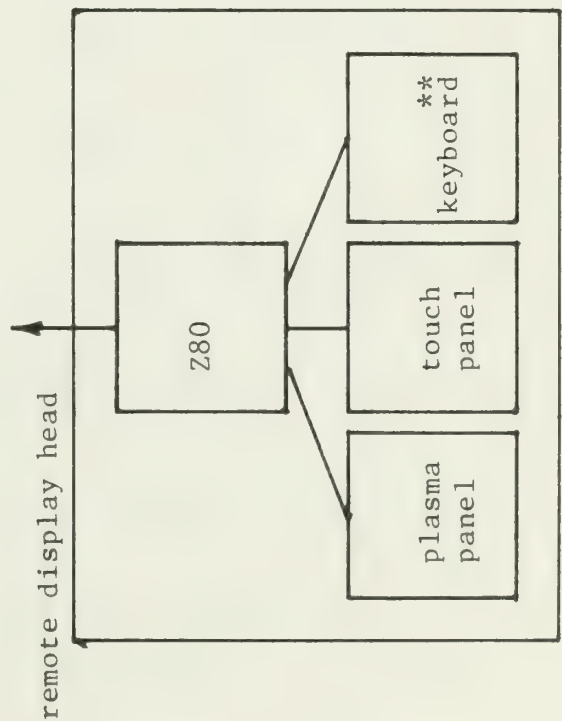
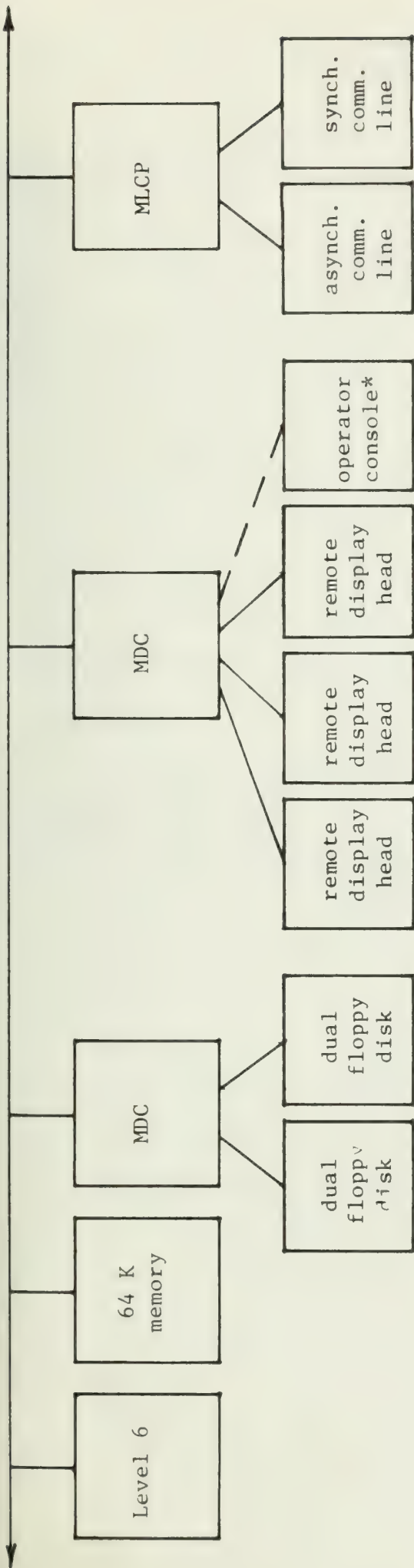
The configuration of this terminal is illustrated in Figure 3. Its components are discussed below.

Processor. This Intelligent Terminal system is based on a Honeywell Information Systems Level 6 model 43 processor. The 6/43 is a member of a new series of Honeywell minicomputers. Some of the more noticeable features of this series are intelligent controllers for each peripheral device and the large amount of directly addressable memory (up to 512K words on the 6/43). Further details of the Level 6 series are available in [6].

Storage. The CAC Level 6 system includes a minimum of 64K words of RAM memory. Two Honeywell dual floppy disk units provide an additional 1 megabyte of removable storage.

Remote Display Head. Communication with the user is performed by three microprocessor controlled remote display heads. Each of these heads have a plasma panel, a touch panel, and a connector for a keyboard. One keyboard, which may be connected to any of the heads, is also included. The plasma panel and touch panel are the same models as those used on the LSI-11 terminal. The keyboard is different, in that it is a standard, unmodified unit. Each remote display head is controlled by a Zilog Z80 microcomputer and is connected to the Level 6 through a standard Multiple Device Controller (MDC) interface. The cables connecting the heads to the Level 6 are 100 feet long so the displays may be located remotely from the system main frame. The details of the remote display head are contained in [5].

Communication Devices. Both synchronous and asynchronous communication capabilities are provided through a Multiple Line Communications Processor (MLCP) interface. This is a programmable interface



*The console is used only for system development

**Each remote display head can support a keyboard. However, only one keyboard is included in the terminal.

Level 6 IT

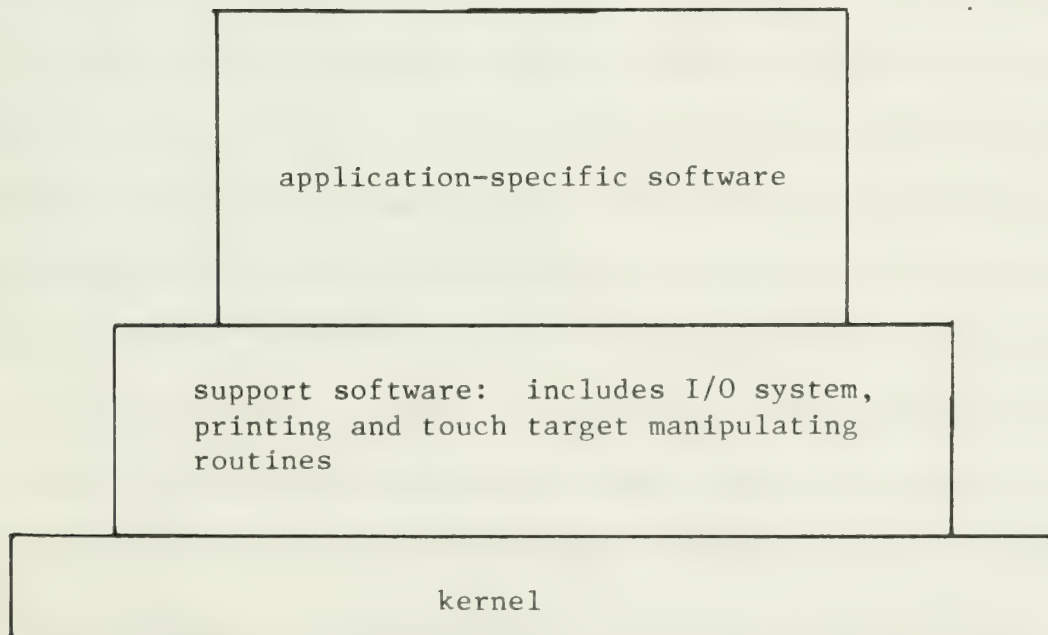
Figure 3

which provides two RS232 asynchronous interfaces as well as two MIL-STD-188C synchronous interfaces.

SYSTEM OVERVIEW

There are three categories of software for the intelligent terminal. The first is the operating system kernel. This consists primarily of process creation, deletion, and synchronization primitives. The second is support software. This includes device drivers, the input-output system, and application-oriented procedures such as graphic display routines and general purpose touch target routines. The third category is the actual application software; this is the "intelligent" portion of the intelligent terminal. The hierarchical structure just outlined is completely conceptual, since there are no hardware or software mechanisms to enforce such distinctions. This structure is diagrammed in Figure 4.

This section describes the first two of these categories: the operating system kernel and support routines. Application software is not described since it varies depending on the application.



Hierarchical structure of IT system code

Figure 4

Operating System Kernel

The operating system uses the disjoint process as the basic functional unit. As implemented by this operating system, a process is simply a procedure (called the process procedure) which has its own stack. The operating system provides a virgin stack to the first procedure executed in each process. Any procedures invoked by the process procedure will use that stack for storing their local data. A benefit of using this method for local storage is that each procedure is reentrant (i.e., more than one invocation of a procedure may be active). Since each process has a separate stack, the address of the stack is used to uniquely identify each process.

Creating a process is very straightforward. First, core for the process stack is obtained from the dynamic memory allocator. This block of core is then formatted to look like a stack, which entails setting five words of control information. Then the stack is modified to appear as if the process procedure had called the scheduler. Finally the stack address is added to the ready queue. When the newly created process is selected for execution, the scheduler "returns" to the first instruction of the process procedure.

Destroying a process is still simpler. All that is required is to free the core occupied by the stack and invoke the scheduler. The scheduler will select some other process to execute, and the old process simply disappears. This mechanism works because there is no way for one process to preempt another, which guarantees the integrity of the old stack until the process has actually gone away. A process can destroy itself either by intentionally calling the process deletion procedure or

by returning from the process procedure, which is equivalent.

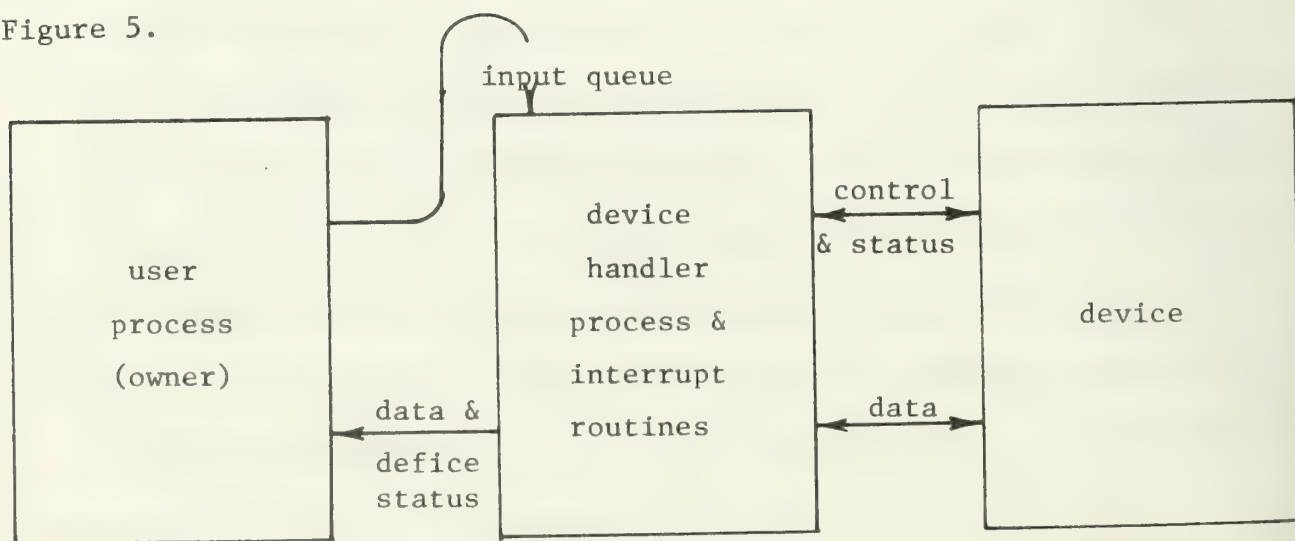
The entire operating system is queue-driven. Each process has an input queue from which it reads its commands, and there is a queue of processes which are ready to execute. Any process which is not on the ready queue is blocked waiting for some event to occur, and is on a queue associated with that event. The mechanism for blocking processes uses the generalized semaphores and P and V operations proposed by Dijkstra [4]. Each semaphore consists of a queue of waiting processes and a count of how many processes are waiting. When a process needs access to some semaphore-controlled resource, it performs the P operation on the semaphore. If the resource is available, the process is allowed to proceed; if it is not available, the process identifier is added to the semaphore's queue and another process is selected for execution. When the process that is currently using the resource is finished with it, it must perform the V operation on the semaphore. If any processes are queued on the semaphore, the first one is removed and placed on the queue of ready processes.

The scheduler is responsible for selecting the next process to run. When invoked, it removes the first process identifier from the ready queue, and makes the stack of that process become the current stack. The priority of each process determines the position in the ready queue of that process' identifier.

The kernel of the operating system is extremely small. The LSI-11 version requires about 800 16-bit words. The entire kernel, with the exception of the scheduler, is written in a high-level language.

Support Software

I/O system. The structure of the I/O system is somewhat unorthodox. Associated with each device is a handler process. Any process which wishes to use the the device must request the handler process to perform the actual operations on the device. The handler process also communicates directly with the interrupt routines for the device. The unorthodoxy of the I/O system stems from a design decision decreeing that only one process may "own" a device. That is, if one process owns a device, any other process wishing to use that device must wait until the first process voluntarily relinquishes it. This decision eliminates the need to multiplex the mass storage device and the communication line to the remote host. However, it also forces these resources to be potentially underutilized since the device will not service the needs of waiting processes even if its owner is not using it. Note that the handler process for a device does not "own" the device. The user process that currently "owns" the device communicates its I/O requests to the handler process, which then performs the device-dependent actions needed to fulfill the user requests. This structure is diagrammed in Figure 5.



How user process accesses device

Figure 5

The I/O system maintains a table which shows the current owner of each device and the handler process associated with the device. When a process requests ownership of a device, this table is inspected to determine if the device is available. If so, the requesting process becomes the owner, and requests from other processes are denied. If the device is unavailable, the requesting process has the option of having the I/O system block him until the device is available or of simply being denied use of the device at this time.

The functions provided by the I/O system are very commonplace. They include reading and writing the device and requesting special services from the handler process as well as requesting and relinquishing ownership of the device. Also included is a routine that indicates how much data the handler process currently has. This is useful since all the other functions of the I/O system are synchronous. That is, a read request does not return until the data is actually returned. Although other processes can execute during that period, the process that requested the I/O service is constrained from running.

Application support. By far the largest portion of code falls into the category of general support software. This includes several routines to operate on the plasma panel, routines to provide formatted output, and routines that allow the creation and manipulation of touch targets. Also included here is the capability to display character sets other than standard ASCII, so the user can display arbitrary patterns of any width and up to 16 dots high.

OPERATING SYSTEM

The IT operating system consists of a number of routines which may be called by user application programs. This chapter will describe those routines which make up the "kernel" of the system. These provide facilities for the controlling of processes. Specifically, they control creation and removal of processes, synchronization and communication between processes using semaphores and queues, and the scheduling of the processes' use of the processor. In addition, there is a memory manager which provides for controlled use of a pool of available memory.

The source for the routines discussed in this section are contained in the subdirectory kernel.

It should be noted that the IT operating system was designed assuming a single user system with correctly operating application programs. Therefore, the operating system is not protected from software errors, or from user-written programs which intentionally try to subvert the operating system. To relax this restriction would require some mechanism such as virtual memory or relocation and bounds registers.

Process Creation, Scheduling, and Removal

Any process running on the IT can spawn other processes using the kernel routine creep. As mentioned in the System Overview, process creation consists simply of allocating a stack for the process, filling in the stack base and dummy subroutine linkage areas, and putting the new process's ID (the address of its stack) into the ready queue.

The newly-created process is in no sense subservient or inferior to its creator. However, the creating process does set the priority of

the new process. In addition, the ID of the new process is returned to the creating process, so the creating process may kill the new one at its discretion.

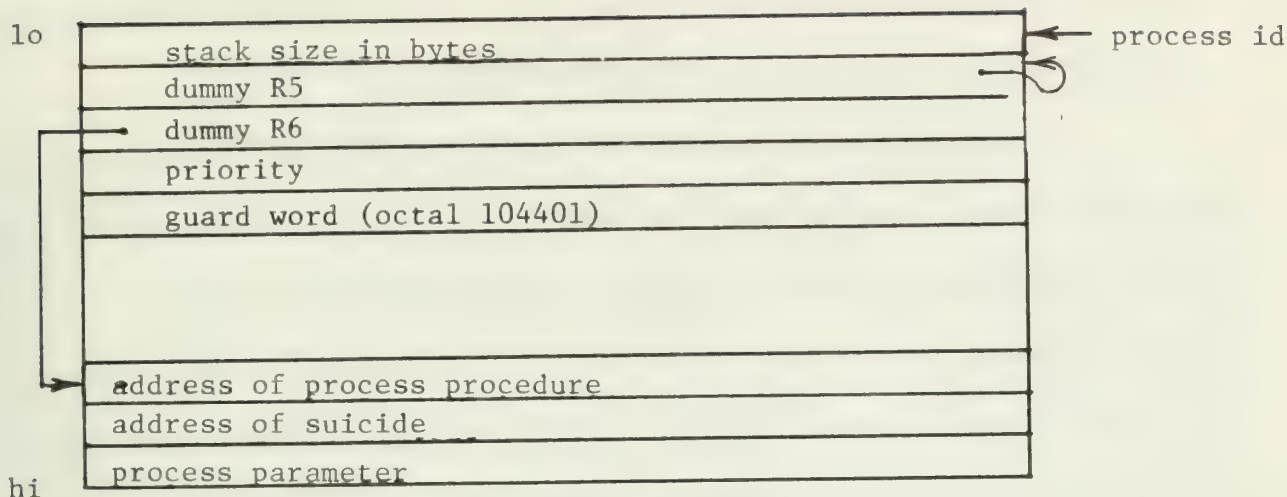
The stack base contains five words used to describe the process and to store specific registers needed to restart a blocked process. The values stored in the stack base differ between the LSI-11 and the Level 6 terminals. For the LSI-11 IT, the stack base contains:

1. the size of the stack, in bytes,
2. two words which will be used by the scheduler to store the process's linkage information and current stack pointer,
3. the priority of the process, and
4. a guard word with the octal value 104401, which is used by the scheduler to check for stack integrity.

A stack base on the Level 6 IT contains:

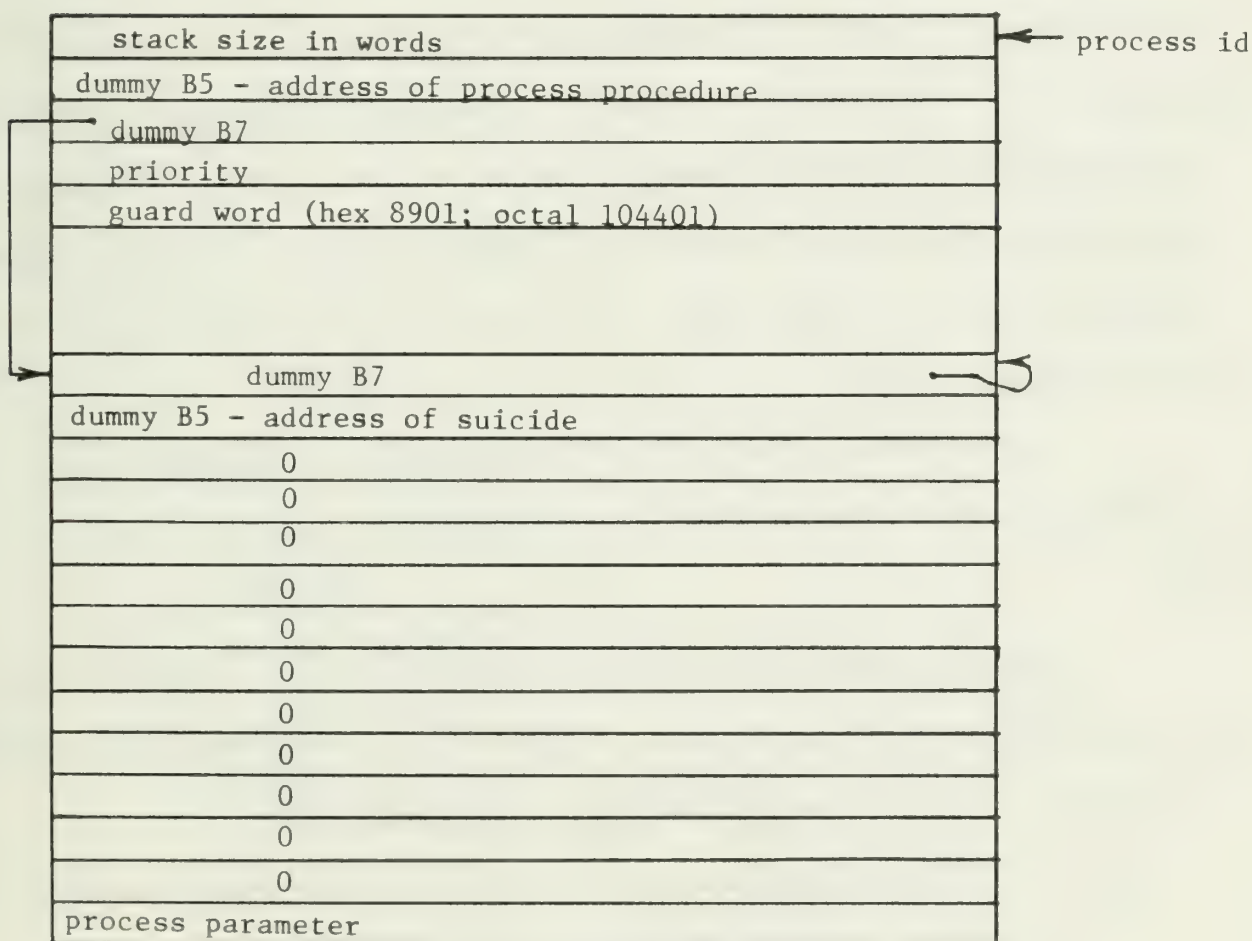
1. the size of the stack in words,
2. two words which will be used by the scheduler to store the return address and linkage information pointer,
3. the priority of the process, and
4. a guard word with value and usage as for the LSI-11 IT.

Creep sets up the stack so that the process, when first scheduled, will start with the procedure whose address is specified as a parameter to creep. Further, creep initializes the stack so that a return by the process's main procedure will invoke the routine suicide. Figures 6 and 7 diagram a process stack as created by creep for the LSI-11 and Level 6 systems. Once created, the new process enters into contention for the



Stack of process initialized by creep on the LSI-11 IT

Figure 6



Stack of process initialized by creep on the Level 6 IT

Figure 7

processor along with all other processes. It has no special privileges or hinderances as a result of being a new process.

A process can go blocked in one of two ways. The usual way is when the process needs to wait for some resource to be freed or for some other process (e.g. a device handler) to do something. This usually happens when the process calls the kernel routine pee, which in turn calls the scheduler routine block. Alternatively, a process can voluntarily go blocked by calling pause. Pause will block the process only if there are processes of a higher priority waiting to run. Otherwise, it lets the current process continue running.

A process can be killed by another process via kill. A process can kill itself either by returning from its main routine or by calling suicide. In either case, the process is killed by freeing the memory used by the process's stack and then entering the scheduler without putting the process on any queue of blocked processes. The process can therefore never be re-awakened.

System Initialization

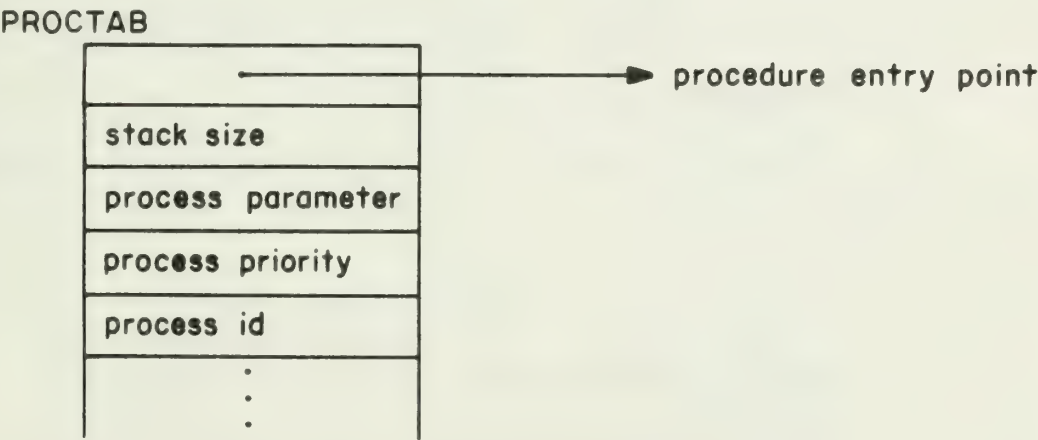
System startup is performed differently on the LSI-11 and Level 6 terminals. The startup function for each terminal is discussed below.

LSI-11 system initialization. System startup is performed by the routine startup which is entered (via a jump instruction at location 0) whenever a system load is completed. Startup will:

1. use routine fixup to determine how much usable memory there is on the machine and clear all unused memory,
2. initialize the I/O system using the routine io init,

- 3. initialize kernel data structures describing the free memory on the system, the pool of queue elements, and the queue of processes ready to run,
- 4. initialize the global variables describing the page, character set and cursor position to use for printing on the plasma panel, and
- 5. create all initially active processes via creep. The user-supplied table PROCTAB contains the entry point, stack size, priority, and one parameter for each initially active process. Figure 8 diagrams one entry in PROCTAB. The entry for startup in Appendix B describes the exact format of this table.

When the processes have been started, startup exits via a call to first block. First block is an alternate entry point into the scheduler. It selects a process to start running without treating startup itself as a calling process. Thus, startup will never be re-entered unless the IT is manually restarted by the user.



Structure of PROCTAB

Figure 8

Level 6 system initialization. When an IT system is loaded into the Level 6 terminal, execution will begin with the procedure entry. Entry is a small section of code written in assembly language rather than a standard procedure. It performs the functions:

1. change the hardware priority level to the level used for application programs,
2. calculate the highest memory address on the machine and point B6 and B7 at this address,
3. push the value of the highest address onto the stack, and
4. branch to startup.

Startup on the Level 6 is similar to, but slightly different from, the LSI-11 version. The Level 6 startup will:

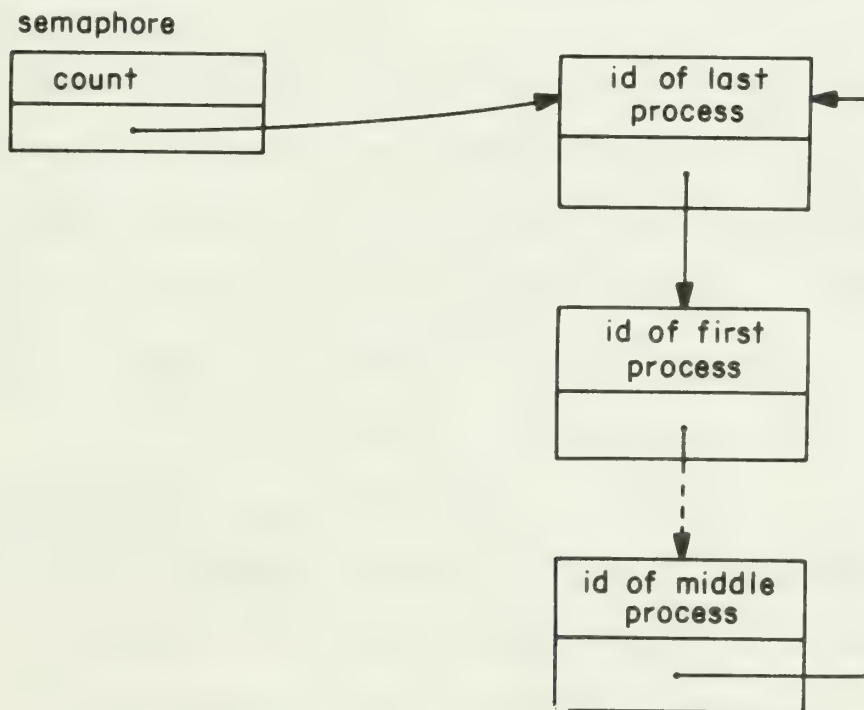
1. disable interrupts from the remote display head controller,
2. initialize kernel data structures describing the free memory on the system, the pool of queue elements, and the queue of processes ready to run,
3. initialize the controlling semaphore for the plasma panel,
4. initialize global variables controlling printing on the plasma panel,
5. create all initially active processes in the same way as on the LSI-11,
6. initialize the I/O system via io init,
7. load the communication controller program, and
8. load and initialize the remote display head controller and allow interrupts from it.

When these functions are complete, startup calls first block, as on the LSI-11.

Process Synchronization

In order for multiple processes to work in harmony, it is necessary to have some kind of synchronization facility to allow one process to wait for another process to complete some task. This facility is provided by the kernel routines pee and vee which respectively implement the P and V primitives described by Dijkstra [3]. These routines operate on structures called semaphores. Semaphores consists of two words. The first word is a count which is either the number of processes which are blocked on that semaphore or the number of outstanding "wakeups". The second word is a pointer to the last element in a circularly linked list. The list contains process ID's for the processes blocked on this semaphore.

When a process needs to wait for some event to occur, it will do a P on the semaphore associated with that event. Pee will subtract



Structure of a semaphore

Figure 9

one from the count. If the result is negative, it means that the process must go blocked, so pee will add the process ID to the list of blocked processes associated with the semaphore and then enter the scheduler to start another process running.

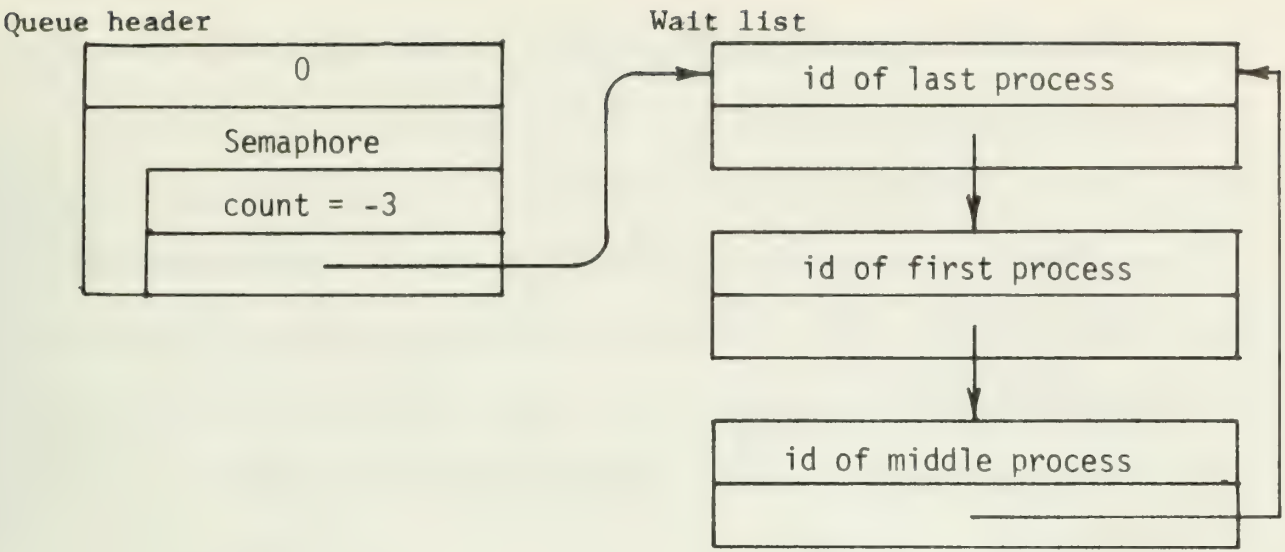
When a process needs to indicate that an event has occurred, it will V the semaphore associated with the event. Vee will add one to the count. If the result is not greater than 0 it means that there is at least one process waiting for the event. In this case, vee will remove one process from the list of blocked processes and put it on the queue of ready processes. This will allow the just unblocked process to contend with other ready processes for access to the processor.

The structure of the list of waiting processes is identical to a standard queue, as described in the next section. This allows the list to be manipulated by the primitives eng and deq.

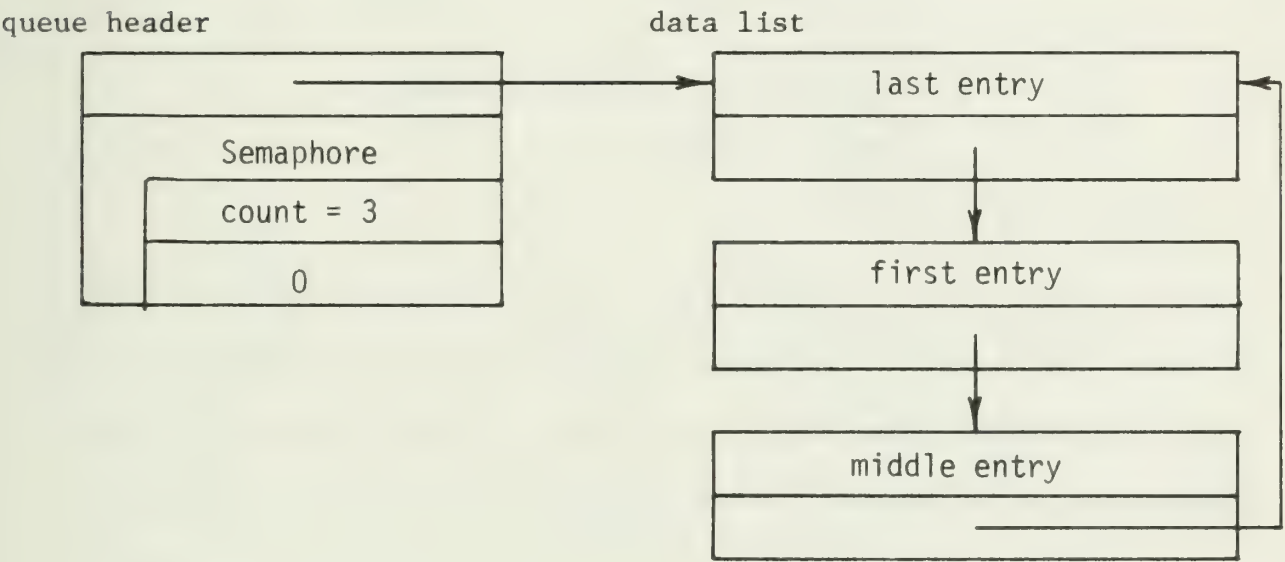
Queues

Queues provide an easy-to-use facility for passing information between processes. Processes can write information one word at a time into a queue, where it can later be read by another process on a first-in, first-out basis. Generally, when more than one word needs to be transferred, it is done by passing the address of a string or structure, rather than using many writes to the queue.

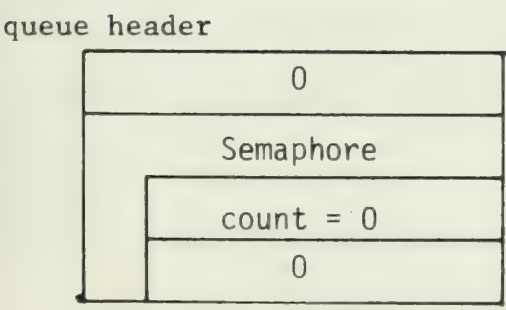
The queue itself is defined by a queue header containing a pointer to the last element in a circularly-linked list of queue elements, and a semaphore associated with the queue. The list of queue elements will contain data values waiting to be read from the queue. The semaphore will control a list of processes waiting to read data from the queue.



A queue with waiting processes



A queue with unread data



An empty queue
Queue structure

Figure 10

Since having both unread data and processes waiting for data to read is an impossible state, at most one of the lists will be in use at any time. Figure 10 diagrams the two states of a queue and an empty queue.

The kernel routines enq and deq are used to add or remove entries from the circularly linked list of queue elements. They also maintain a pool of free queue elements. User programs should access queues using the routines read q and write q explained below.

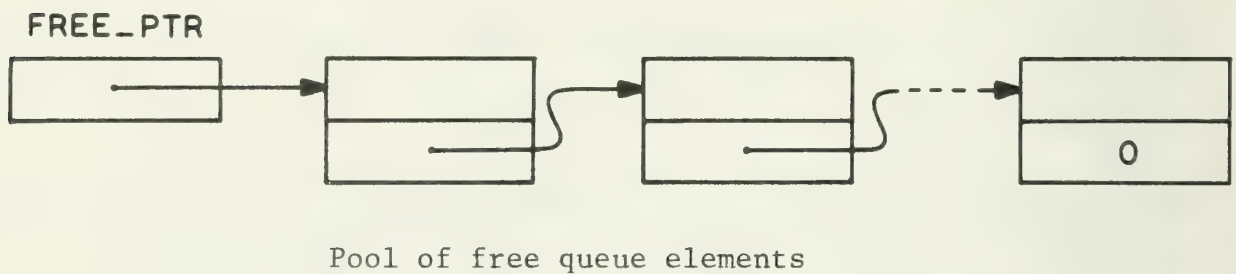


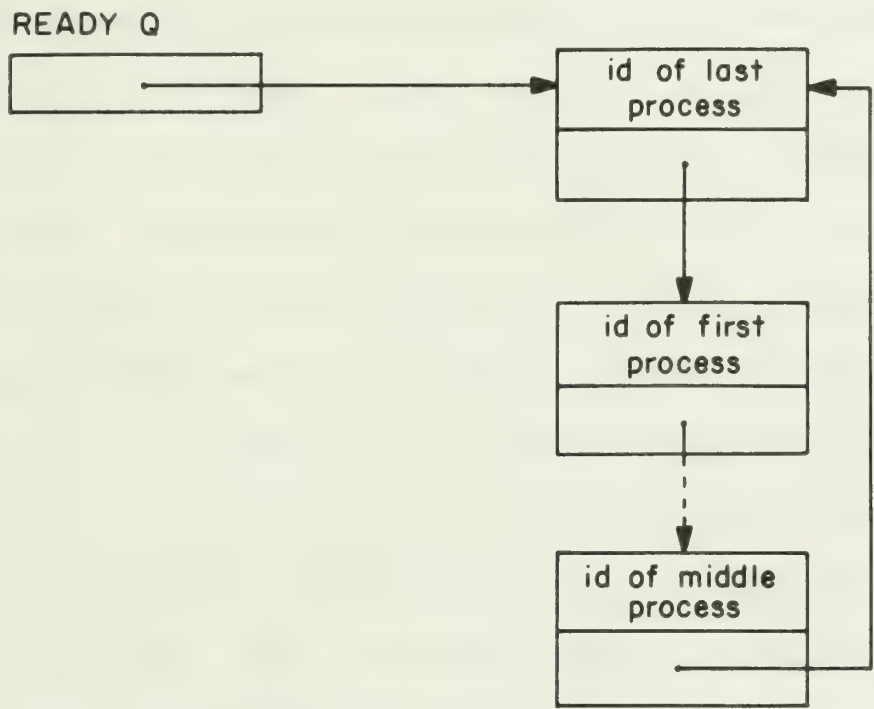
Figure 11

When a process wants to enter a word into a queue, it calls write q. Write q will use enq to add the word to the queue. Then, it will V the queue's semaphore. If there is a process waiting to read from the queue, the V will allow the reading process to start up.

When a process wants to get information out of a queue, it calls read q. Read q will first P the semaphore controlling the queue. If there is data in the queue, pee will return immediately. If there is not, pee will cause the process to go blocked until another process writes to the queue. When pee returns, read q will use deq to remove one entry from the queue.

The system ready queue has a slightly different format from a normal queue. There is no semaphore associated with the ready queue.

Also, it is not a strictly first-in, first-out queue, but rather the elements are sorted by process priority. The routine enq RQ is used to add elements to the ready queue to maintain the ordering by priority. Process ids are added to the ready queue so that processes with higher numbered priorities are serviced before those with lower numbered priorities. Within each priority, the ready processes are given control of the processor on a first-in, first-out basis.



Structure of queue of ready processes

Figure 12

Scheduler

The scheduler is an integral part of the routine block, which is called whenever the currently running process must go blocked. After putting the current process to sleep, block will select the next process from the ready queue and start it running. If there are no ready processes, then block will stop until some process becomes ready. On the LSI-11

IT, this is done using the WAIT instruction. On the Level 6, block enters a busy loop to wait for some process to become ready. Usually, this will happen when an interrupt occurs and the interrupt handler writes to a queue or V's a semaphore.

Memory Management

The routines alloc and free can be used to dynamically obtain and release pieces of memory. The table CORETAB contains two word entries describing the size and address of each currently unused piece of memory. Alloc will select a piece of memory with the specified size from CORETAB using a first-fit algorithm. Free can be used to return a piece of memory back to the free pool. Any integral number of words may be allocated or freed. No check is made to ensure that memory which is being freed is currently unused. It is the caller's responsibility to use alloc and free properly.

Interrupts and Traps

The LSI-11 and the Level 6 handle interrupts and traps differently. The mechanisms for both machines are described below.

LSI-11 mechanism. Due to the way the LSI-11 hardware handles interrupts, the lowest 400 (octal) bytes of memory must be set aside for interrupt vectors. The assembly language source in low.s is used to set these up properly. The first two words defined in low.s contain a branch to the routine startup. This is used when the system is started up. The rest of low.s contains interrupt vectors. When an interrupt from a known device occurs, the process status word will be loaded with a number whose low-order four bits are an IT device type code, and the system will branch to the routine int disp. An interrupt from an unknown device will cause a branch to a halt instruction.

All valid interrupts go thru the routine int disp. This routine uses the device-type code, obtained from the low-order four bits of the process status word to index into tables of interrupt handlers and priorities. These tables are used to select which routine should handle the interrupt, and the priority at which the routine should run. This scheme allows for a maximum of sixteen devices to be attached to the IT. When adding and/or deleting devices or device types, it is necessary to change the interrupt vectors in low.s as well as the tables internal to int disp.

Traps all go to the trap handler called trap. There is no valid way to handle traps in the IT system, so any trap is a fatal error, causing the entire system to halt.

Level 6 mechanism. The Level 6 assigns special meanings to the values stored in the lowest 100 (hex) words of memory. The assembly language source in low.a is used to set these up properly. Low will

1. initialize the trap save area,
2. initialize the interrupt save areas for IT devices, and
3. zero areas not used.

When an interrupt occurs at any given hardware level, the hardware will:

1. pick up the interrupt vector for the current level,
2. use this as a pointer to an interrupt save area where the current registers and the PC are stored,
3. pick up the interrupt vector for the interrupting level,
4. use this as a pointer to an interrupt save area from which the registers and PC are loaded, and
5. continue execution at the new level with the new PC

Level 6 IT systems are set up so that all interrupts will be handled by

the procedure levlp. All the registers are restored from the interrupt save area. Two registers have particular values:

1. B6 points to the stack for the interrupt procedure, and
2. B3 contains the address of the interrupt procedure.

Levlp merely calls the procedure indicated by B3, and then does a LEV when it returns to cause processing to continue at the next ready level.

To accommodate a new device, low.a must be modified to provide an interrupt vector, an interrupt save area, and an interrupt stack for the device. It should be noted that levlp, unlike int disp, does not need to be modified when devices are added to the system.

There is no valid way to handle traps in the IT system, so any trap is a fatal error.

Errors

When any routine in the system encounters a fatal error, it calls the routine error. Error prints a message on the plasma panel and then halts. The processor can be restarted at this point using the operator console, but the results are (usually) unpredictable.

C Program Support

There are a number of special procedures to supply assembly language functions to C programs. Specifically, these are mfps, mtps, and halt. In addition, two procedures ldiv and lrem allow the user to get the results of doing division on a double word dividend.

I/O SYSTEM AND DEVICE HANDLERS

The general structure of the I/O system of the IT is described in the section System Overview. This section discusses the implementation of the I/O system in more detail. The structure of standard I/O routines is presented, followed by a description of the processes which handle the various system devices. The final subsection describes the naming conventions used by the IT for referencing devices and disk files.

The source files for the routines directly connected with this discussion are contained in the subdirectories devices, io sys, and disk.

I/O System

The IT I/O system provides several functions for accessing either specific system devices or one or more disk files. Most of the procedures provide common functions for accessing these resources; some are less obvious. The functions which access a single resource are:

1. open - obtain ownership of a resource,
2. close - relinquish ownership of a resource,
3. read - input data from a device or file,
4. write - output data to a device or file,
5. peek - get the number of a device,
6. flush - remove any unread data from the device handler process's buffer,
7. seek - position the read/write pointer in a file,
8. set_mode - a "black hole" function that allows application programs to communicate in non-standard ways with device handlers,

9. create - create a new disk file, and
10. delete - remove a disk file.

There are two additional functions available:

1. io_init - initialize the I/O system, and
2. clear_io - cleans up any outstanding I/O requests by the current process.

These last two procedures are rather straightforward and are described in Appendix B.

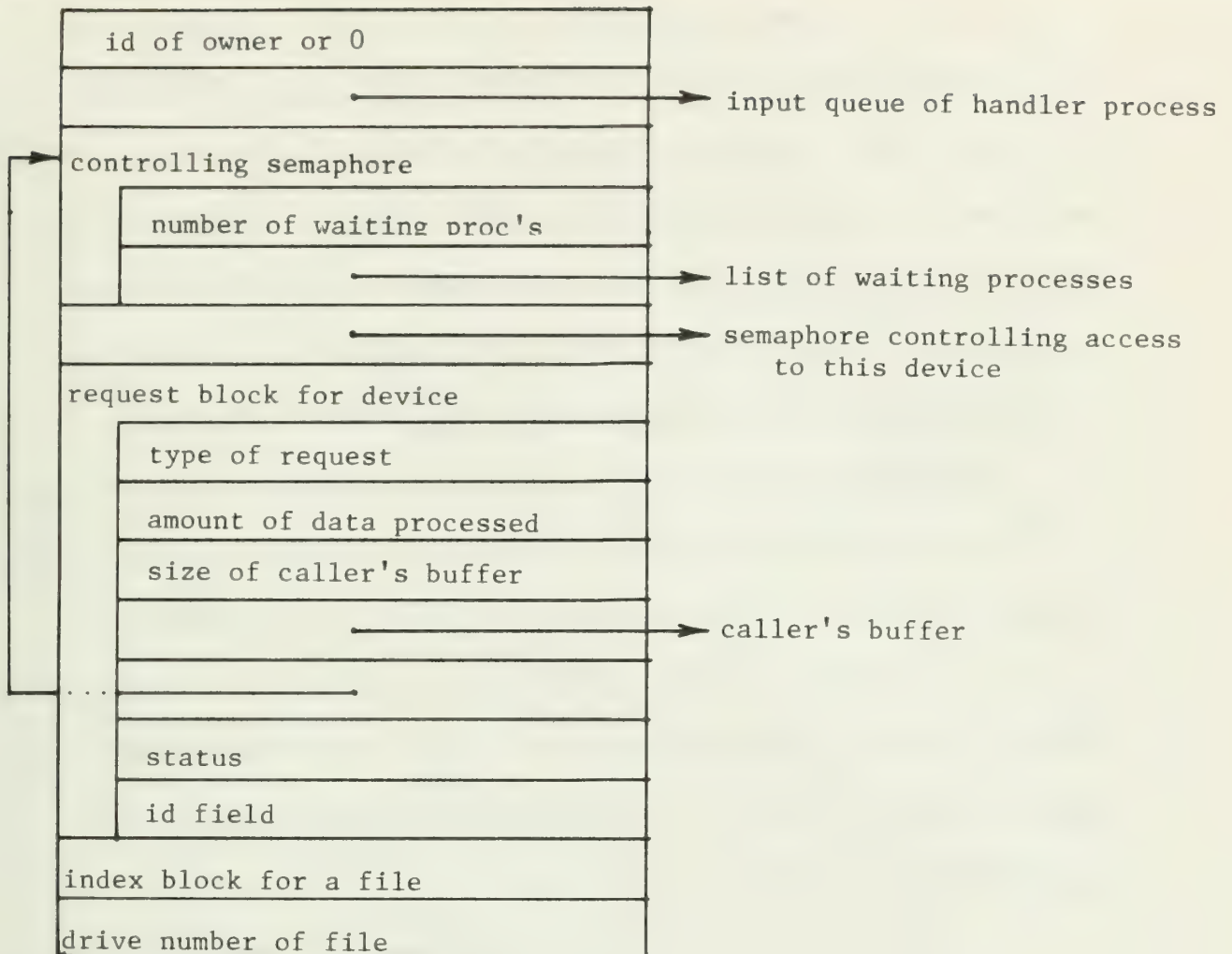
All of the functions which access a single device or file, except open and create, are very similar in structure. Each of these procedures performs the following actions:

1. verify that the caller has specified a valid device or file identifier,
2. check the entry in the system device table DEV TAB to verify that the calling process owns the resource,
3. format a request block for this action,
4. send the request to the handler for the resource,
5. P the semaphore in the request block, and
6. set caller's return status and return.

The handler for the specified device will actually perform the work associated with the request. The handler must V the request semaphore when the task is completed so that the requesting process can continue.

The open routine follows a slightly different format. Initially it calls determine to convert to an integer the name of the resource being opened. The integer is used to index into the system device table. Open then checks to see whether the device or file is unowned or if the caller owns it. If both of these tests fail, then the caller has the

dev_entry



Structure of DEV_TAB entry

Figure 13

option of waiting for the resource or having an error code returned. If the resource is available to the caller, then open will:

1. mark the resource as owned by the caller,
2. send a flush request to the handler,
3. send an open request to the handler, and
4. return the id of the opened resource.

If any errors are encountered, the resource is closed and an error code is returned.

The procedure create has a drastically different format. Create uses other I/O procedures such as open, close, and read to create a new disk file. Initially create tries to open the file. If that succeeds, the file is truncated to zero length and create returns. Otherwise create opens the parent directory of the file and adds an entry for the file. It then creates the zero length file on disk. Finally the new file is opened and create returns.

Each of these I/O procedures is explained further in Appendix B.

Notes. More detailed information on the structure of the I/O system is included in the section Modifying the I/O System.

It should be noted that, for better or worse, the plasma panel is not treated as a standard device. The I/O routines do not access the panel, rather each process can perform operations on it directly.

Device Handlers

Purpose. Every device in an IT system, other than the plasma panel, has its own device handler. Device handlers are used to perform the device-specific tasks for each I/O function and to buffer incoming data until it is read. Each handler runs as a queue-driven process, receiving commands from the interrupt routines for the device and from the I/O routines. Current IT software supports four devices:

1. keyboard,
2. touch panel,
3. asynchronous communication line, and
4. synchronous communication line.

In addition, named files resident on floppy disks are also supported. The section on Application Software explains how to access the plasma panel.

Structure. All the device handlers are very similar in structure. Each of them perform some amount of initialization then enter a never-ending loop. Within the loop, the handler:

1. reads from its input queue. The handler process will block at this point until a request is sent to it.
2. interprets the data from the queue. If the command is from an I/O system routine, the data will be a pointer to a request block describing a user's request for an I/O operation. Commands from the interrupt routines are directly encoded in the data.
3. acts on the command. The action taken will depend on the specific command. Types of commands include all the standard I/O functions plus receiving data, noting that the device has finished a write, and noting whether the device is currently ready for use.

When the handler finishes some action for an I/O routine, such as reading or writing data, the handler will V the semaphore associated with the request. Since the I/O routine P'd the semaphore after initiating the request, this action restarts the requesting process.

Interrupt Routines. The devices of the IT communicate with their handling processes by interrupt routines. Since the LSI-11 and the Level 6 have different interrupt mechanisms and device interfaces, the low level communication between a handler and its device are very different on the two systems. Each approach is described below.

LSI-11: The LSI-11 can access the registers for its devices directly, in the same manner it accesses memory locations. This affects the workings of the device process as well as the interrupt routines.

The input interrupt routine for each device gets its data directly from the device registers. The data is then encoded along with a flag to the handler, and then sent to the handler via write_q.

Input data is stored by each handler in an internal cyclic buffer. As data is read by other processes, it is deleted from the local buffer. Any data input when the buffer is full is dropped, typically with no indication that data has been lost. The entries in Appendix B for the handlers (kbd driver, ph driver, etc.) give more information on the exact results of reading from each device.

The device handler communicates with its output interrupt routine via global variables. To initiate an output, the handler sets a global count and buffer pointer, and then invokes the output interrupt routine by enabling output interrupts for this device. When the interrupt routine has transferred the entire buffer, it sends a special command to the device handler which then disables output interrupts.

Level 6: The Level 6 cannot access the registers for its devices directly, rather it must use special IO instructions. This affects the interaction between the handler, the interrupt routines and the device. Also, unlike the LSI-11, Level 6 devices operate in a direct memory access (DMA) mode when inputting or outputting data.

When an input interrupt routine is invoked, it copies the data stored in its buffer by the device to the handler process's data buffer. If the handler's buffer was empty, the interrupt routine sends a message to the handler telling it of the new data. If there is not enough room in the buffer for all the data, that data is discarded and a buffer overflow message is sent to the handler. The DMA capabilities of the device are not used to input the data directly into the handler's buffer

because:

1. it is a cyclic buffer, and the devices will not handle the wrap around condition, and
2. it is not possible for the device to detect, and avoid, potential buffer overflow.

On output, the process handler itself initiates the transfer.

The output interrupt routine is invoked only after the entire transmission is complete. At that point the interrupt routine sends a command to the handler which then disables output.

Although at the lower levels the device interfaces on the LSI-11 and Level 6 terminals are very different, to the user application they appear to act the same. Thus the comments explaining the details of the results of reading from or writing to a device, as explained in Appendix B, are the same for both systems.

Naming Conventions

In order to initially obtain ownership of a device or file, a process must open or create that resource. These two procedures each take a parameter which specifies the resource to be owned. To insure access to the correct resource, the process must specify the resource name exactly as it is known by the I/O system. This subsection briefly describes the names recognized.

There are two types of names accepted by the I/O system: names of devices and names of files. Both types of names are represented by standard C-language character strings. That is a string of ASCII characters terminated by an ASCII NUL (octal 0). Any ASCII character except NUL is valid in a name, but "/" (slash, octal 57) has a special meaning, described below.

Device Names. The names of the devices on an IT are defined in the file constants.incl and are descriptive of the device with which they are associated. The names and their defined values are:

1. KEYBOARD: `"/dev_kb"`,
2. TOUCH_PANEL: `"/dev_tp"`,
3. PHONE: `"/dev_ph"`,
4. VIP: `"/dev_vip"`,
5. DISK0: `"/dev_disk0"`,
6. DISK1: `"/dev_disk1"`,
7. DISK2: `"/dev_disk2"`, and
8. DISK3: `"/dev_disk3"`.

Note that DISK2 and DISK3 are defined only on Level 6 systems.

No other names are recognized as referencing a system device.

File Names. Any name not recognized as a device name is interpreted as a file name. File names are composed of two parts:

1. the name of the disk containing the file, and
2. the name of the file.

The name of the disk must be preceeded and followed by a slash ("`/`", octal 57) which serves to separate the parts of the file name. For example, the name of file "A" on disk "data_3" is `"/data_3/A"`. The name of the disk containing the file can be specified in either of two ways:

1. by the name given the disk when it was initialized, e.g.
`"data_3"`, or
2. by the name of the drive on which the disk is mounted, e.g.
`"dev_disk0"`.

In the above example, if the disk named "data_3" is mounted on drive 1, then `"/dev_disk1/A"` is completely equivalent to `"/data_3/A"`.

In addition to the file names just described, the IT software supports a shorthand method for designating files which reside on the floppy disk mounted on drive 0. Such files can be referenced by the name of the file only, without specifying the name of the containing disk. Under this convention, if the disk "data_3" from the previous example is mounted on drive 0, then file A can be referenced simply as "A". File names which start with any character other than / are interpreted as such shorthand names, and will be looked for or created on drive 0. Shorthand names, e.g. "A", are completely equivalent to complete names with a disk name of "dev_disk0", e.g. "/dev_disk0/A".

Note that the IT procedures which access disk files contain most of the code necessary to support a tree-structured directory hierarchy of files, complete with the concept of a user-specified working directory. Future IT programmers may find the implementation to a multiple directory hierarchy to be a simple and useful expansion.

Drive 0

disk:	data_3
files:	
	a
	b
	c

Drive 1

disk:	janus_demo
files:	
	x
	y
	c

Examples of equivilent names:

`"/data_3/a" ≡ "/dev_disk0/a" ≡ "a"`

`"/janus_demo/x" ≡ "/dev_disk1/x"`

`"c" ≡ "/data_3/c" ≡ "/dev_disk0/c"`

Note that "c" does not refer to the file c on janus_demo since it is mounted on drive 1.

File naming conventions

Figure 14

APPLICATION SUPPORT SOFTWARE

The largest single group of code for the IT is the general support software. This includes routines for performing graphics and printing on the plasma panel, for manipulating touch targets, for manipulating character strings, and for displaying data. Each of these types of routines is discussed below. The source code for these routines is found in the subdirectories panel, print, tt, strings, and display.

Plasma Panel

The LSI-11 IT has a single plasma panel display, and all panel accessing routines affect this display. The Level 6 terminal, however, may have up to three plasma panels - one in each remote display head. This leads to the need to specify which panel(s) are to be affected by a particular action. When a Level 6 system is started, it will perform the necessary set up so that panel 0 is initially the only plasma panel selected. All panel accessing routines will affect all selected panels. If the user wishes to access a different set of plasma panels, the procedure set pnl can be used. This allows any combination of plasma panels to be selected for subsequent accessing. Future use of panel functions will affect all the plasma panels selected by set pnl. This will be true until another call to set pnl changes the combination of panels selected.

When accessing the plasma panel there are two types of functions available: graphics and printing. Each of these types of functions is described below.

Graphics. The graphics routines for the panel are very basic. They allow the caller to light or erase:

1. a single dot,
2. a masked vector 16 dots high,
3. a straight line, or
4. all of the dots within a specified rectangular area.

Additionally, all the dots on the panel can be turned off at once.

On the LSI-11 IT, the panel graphic routines access the registers in the plasma panel interface directly. Appendix E is a detailed discussion of the meanings of the various register values for this interface.

It should be noted that the map display routines include code to display an arbitrary collection of closed polygons and to shade the interiors of these polygons. However the map routines will need to be extensively revised before they will be generally useful.

Printing. More IT routines are associated with printing on the plasma panel than with any other single activity, with the possible exception of the application-specific code. The areas of activity include:

1. page specification,
2. character set specification,
3. cursor position,
4. printing environment,
5. simple printing, and
6. tokenized printing.

Each of these areas will be briefly explained below.

Page specification: For printing purposes the plasma panel can be logically partitioned into any rectangular sub-portion. The default page is the entire panel, but the user can set it to any size. The origin of the page, its lower-left corner, can be any place on the

panel, but the page will always be oriented parallel to the edges of the panel. The page size and position can be set or determined by application programs.

Character set specifications: A variety of character sets can be used for printing on the plasma panel. The standard character set has true upper and lower case characters, and uses a 16x8 dot area for each character. However, as all characters are drawn on the panel by software, application programs can use the standard print routines to display other sizes of characters simply by changing the character set used. User-defined "characters" may in fact consist of specialized graphics.

A character set is specified by a cs_desc structure (see Appendix C) and two sets of vectors. Application programmers can create their own special purpose character sets by the following steps:

1. Create and fill in a cs_desc structure. Note that characters may have any width but cannot be over 16 dots high.
2. Create an array of bytes which is the effector table for the charset. An entry of 0 indicates that the graphic for the corresponding character is displayed on the plasma panel when the character is encountered. Other values indicate format effectors whose actions are described in Appendix D.
3. Create the integer array of masks corresponding to the new characters. This array is of the form:

masks [number_of_chars_in_charset][width_of_each_char].

When character *i* is to be printed using this character set, the width_of_each_char vectors starting with masks [*i*][0] will be displayed on adjacent vectors of the plasma panel via put.

The dots which correspond to 1's in the masks will be turned on to display the figure.

The application can then use the new character set by calling set charset with a pointer to the new cs desc structure.

Several routines require parameters or return values specified in character sizes. These values, which are generally stored internally in dots, are interpreted using the size of characters in the current character set. If the character size changes, by switching character sets, the corresponding values may also change.

Cursor position: The position on the page where the next character is to be printed is determined by the value of the internal cursor. This value is stored internally as a pair of dot coordinates which are relative to the current page definition. If the page is changed, the cursor position on the panel will change accordingly. However if the character size changes, printing will continue on the same line as before using the new character size. The cursor position can be updated explicitly by the application program or implicitly by printing on the panel. Appendix D indicates the effect each character in the standard character set has on the cursor position.

Printing environment: The page specification, the character set identifier, the cursor position, and the set of selected plasma panels (for the Level 6 IT) taken together determine the current printing environment of the IT. Since there is only one system-wide value for each of these attributes, changing any of them in one process or procedure will change that attribute for every process and procedure in the system. As a result it is generally a good idea for each routine that alters the printing environment either explicitly or implicitly to save the existing

environment and to restore that environment when it is finished. Two routines, get env and set env, make this easy to do.

Simple printing: Formatted output to the plasma panel can be done via printf. The IT version of printf is similar to the UNIX version with restrictions on the format specifications. Printf will cause the formatted output string to be printed on the plasma panel within the current page. Printing will move the cursor according to the effect of the character printed. After each character, the cursor position is checked to ensure that it is still within the specified page area. If it is outside of the area horizontally, the cursor is moved down a row and to the left edge of the page. If it is out of range vertically, the cursor is moved to the top line on the page. This line and page wrapping occur whenever the cursor is determined to be out of bounds regardless of the characters being printed, so that wrapping will generally break words.

Tokenized printing: To avoid the problems of words being broken at page boundaries, tok print will do tokenized printing. This routine is similar to printf except that it breaks the formatted string into words, or tokens, delimited by one of a user-supplied set of characters. Each token is printed and followed by a user-supplied separating string. Before the token and separator are printed, their combined length is compared to the remaining space on the current line. If they will not both fit, the cursor is moved down a line before they are printed.

Level 6 Interface. On the Level 6 terminal, the plasma panel and the touch panel are part of the remote display head. In this system, all communication with these devices is performed through a microprocessor which controls the display. As a result, the low level software routines

to access the panel on the Level 6 are significantly different from those on the LSI-11. The details of these procedures are described in the section Accessing Remote Display Head. However, the support routines mentioned in this section, e.g. put, erase, etc., are called and used exactly the same on both systems.

Touch Targets

Routines have been built for the IT to facilitate the use of the touch panel for user input. These routines manipulate entities referred to as "touch targets" or simply "targets". In the minds of most programmers of the IT, touch targets are linked closely not only with the touch panel but also with the plasma panel. This association should be explained before proceeding so that later comments can be better understood.

Physically the touch panel is a completely separate device from the plasma panel. However, it is placed in front of the plasma panel, so that the two surfaces are superimposed and to the casual observer appear to be one. When a user touches the glass of the plasma panel, his finger will interrupt a pair of intersecting infra-red light beams from the touch panel. The coordinates of the interrupted beams are passed to the IT software and constitute a "touch". The coordinates of the touch will then be transformed by the software into some internally meaningful value. The utility of the touch panel is obviously greatly increased if touches to different portions of the touch panel have different meanings for the software. To do this, there must be some mechanism by which the IT system can convey to the user the meanings of various areas of the touch panel. The IT system uses plasma panel graphics in positions correlating to meaningful areas of the touch

panel. For example, an application may display a rectangular box on the plasma panel and then wait until the user breaks the touch panel beams in front of the box before taking some action. Alternatively, the application may wait for the user to break the light beams and then display a marker on the plasma directly behind where the interrupted beams intersect. Since the light beams of the touch panel are invisible, it appears to the user of this application as if touching the plasma panel has caused the marker to be drawn. The user tends to ignore the touch panel and think only of displays and touches on the plasma panel as being meaningful. As a result there has been a strong tendency on the part of IT programmers to closely associate meaningful areas of the touch panel with a corresponding plasma panel graphic. Both a specific area of the touch panel and its corresponding graphic are involved in the concept of a touch "target".

The following discussion describes general attributes of touch targets and the types of functions which can be performed on them. The source for the routines mentioned are found in the subdirectory tt.

General structure. On the IT, a touch target is a distinguished area of both the touch panel and the plasma panel. The area is rectangular, but several targets may be combined to form irregularly shaped targets. Each target has a user-supplied value associated with it. This can be any value which can be represented in one 16-bit word. It is supplied for the use of application routines. When user touches are "read" through the standard target routines, this is the value that is returned to the calling program. The application program can use this value to determine what action to take next.

In addition to a size, position, and value, each target may

optionally have a text label and a rectangular outline. If these standard graphics are not used, the application may use its own graphics to represent touch targets. Finally, associated with each target is a collection of flags which determine attributes of the target. The most significant of these flags are the three which determine the remote display head(s) for which the target is defined. A target can be defined on any combination display heads. If it is defined for more than one, the graphic for the target will be displayed on multiple plasma panels and a touch of any of the displays will register as a touch to this target. These flags are further explained in the entry for tt create in Appendix B.

Manipulating Touch Targets. The most commonly used functions for manipulating a single target include:

1. creating it by filling in a target structure,
2. activating so that user touches to this target will be recognized,
3. erasing an activated target, and
4. deleting an active target.

Activating and deleting a target are asymmetric actions. Activating a target includes displaying it if it uses the standard graphics. Deletion does not erase the target, but merely removes it from the list of active targets. This difference was implemented primarily due to speed considerations and the way in which targets are typically used in the IT systems produced to date. In these systems, individual targets were seldom deactivated without deleting all the currently active targets and displaying an entirely new page. As a result, it is much faster to erase the entire screen at once than to erase each target individually. The procedure tt deactivate can be used

to erase and deactivate a single target.

A few routines exist to manipulate groups of touch targets:

1. A group of targets in adjacent spaces and of the same size will be created by tt arranger.
2. A number of user touches will be read and the values of the touched targets returned by tt selections.
3. All the active targets can be deleted by tt cleanup.

The routine tt selections is designed to be the standard method for reading user touches. Application programs can also read touches one at a time directly via tt read. However tt read does not return the value of the touched target. Rather, for internal consistency, tt read returns the index into the system array of active targets, tt current, of the touched target. Application programs which call tt read directly will need to do some calculations on their own to determine the value of the touched target.

Other touch target routines exist. These are less generally applicable and include functions such as:

1. draw the outline of a target,
2. print the label for a target,
3. light or erase all the dots in the area of the target,
4. distinguish a target by a small mark,
5. reposition a target without changing its other attributes, and
6. change the label on a target.

All of these functions, except the last one, work only on activated targets.

Notes. The touch target routines assume that the application routines do not change the values in the system array tt current. If

an application routine does modify tt current, attempts to use any procedure which accesses this array may result in an illegal value being used as a pointer to a target structure.

The label for a target is stored in the target structure as a pointer to a character string rather than as a copy of the string. If the contents of the string are changed after the target is created, then the label will also change.

String Manipulation

Since the C language has no built-in string manipulating functions, the IT software includes routines to perform some of the more common functions. These functions include:

1. searching the contents of a string,
2. parsing a string, and
3. formatting and converting strings.

All of these routines, except as noted, follow the C convention of expecting character strings to be null-terminated. Appendix B contains entries for each of these routines explaining in detail how to use them. The files containing source code for these routines are in the subdirectories strings and print.

Searching Strings. The PL/1 functions index and verify were implemented to aid in scanning the contents of a string. Index matches one string to a substring of another. Verify finds the first character of a string that is not in a second string.

Two strings or parts of strings can be compared for equality by cmp. Cmp is somewhat different than other string functions since it takes the length of the strings to be compared as a parameter rather than acting on everything up to the NUL. This can be used to compare

substrings of one or both strings.

String Parsing. A string can be broken into tokens by repeated use of get token. Get token takes a string to be parsed and a set of delimiter characters. The returned token string will be the first substring of the parsed string which consists of non-delimiters and is surrounded by delimiters.

String Formatting and Conversion. Character strings can be expanded according to a format specification by ln xpand. Ln xpand is similar to printf in its interpretation of format specification and parameter replacement. However, instead of printing the string on the plasma panel, ln xpand returns the expanded null-terminated string to the caller. This function is very useful for expanding a string before passing it to some routine such as write which does not do parameter expansion.

Character strings representing numbers can be converted to the internal binary format, and vice versa. Converting strings to binary is done by cvb. This routine will handle octal or decimal numbers. It follows the standard C convention of using a leading 0 to specify octal numbers.

Expanding a binary number to its corresponding character string can be done indirectly by ln xpand or directly by parm xpand. However, the parameters to parm xpand are rather awkward as it is geared to work with ln xpand. The easiest way to convert a number to its string is to call ln xpand with a format of octal, decimal, or hexadecimal as desired.

Data Display

This section discusses the IT routines that display data items

in various formats. A word of explanation is needed before proceeding, however. These routines were created as part of a demonstration system designed to test the feasibility of the IT concept. As a result, they are very closely tied to that initial application system. There are several shortcomings in these routines:

1. they make assumptions about the area of the panel that can be used for display,
2. they assume that all characters use the 16x8 grid,
3. the data value -32765 is interpreted as a special code indicating missing data, and
4. their internal organization is poor.

The code for these routines will need much revision before they can be generally useful. They are included primarily as an example of one way to do things.

These routines described here will display data in three different formats:

1. tabular,
2. bar chart, and
3. shaded map.

The three types of display are described below, noting the outstanding shortcomings of each. The files containing source code for these routines are in the subdirectory display.

For the remainder of this discussion, the following definitions will be followed:

1. An "item" or "data item" is an array of logically related numbers, for example the populations of the counties of Illinois
2. An "element" is one of the entries in an item, for example the

population of Cook county.

3. A "value" is the numeric value of an element.

Table. The tabular display function is the most general of the data display routines. It takes from one to three data items of not more than twenty-nine elements, and the labels for each row and column of the display. From this information it produces a horizontally and vertically centered table of the data. The limited amount of data that can be shown results from the use of 16x8 characters. If table is modified to use the variable character sizes provided in the IT system, more data could be displayed.

The only known bug in table is the assumption that each character takes 16x8 dots. If table is called using a different size of characters, the spacing of the resulting display will be incorrect.

Bar graph. This routine will display up to twenty-nine elements of a data item as a bar chart, labeling each bar and the entire chart with caller-supplied labels. The caller specifies the area of the panel to be used. The chart will be centered vertically in this space and will use the entire width.

Bar graph has one known bug: if the difference between the largest and the smallest values in an item is large enough to cause overflow, no bars are drawn for the elements in that item. This bug is directly related to the scaling algorithm. The scaling algorithm determines how wide to make the bar for each element, based on the value of that element and the range of values in the entire chart. The algorithm sets the minimum value to relative zero, then scales all other values linearly above this. Problems with this algorithm include:

1. The absolute magnitude of the values are not obvious. For

example, an item with values 1, 2, 3 will produce a graph of the same size and shape as one with values 1001, 1002, 1003.

2. Values less than zero in an item cause the zero axis of the graph to be to the right of displayed axis. This is misleading

In addition to the problems related to the scaling algorithm, bar_graph has several other shortcomings. In particular:

1. The assumption of 16x8 characters permeates its internal calculations.
2. It calls put ascii to print the bar labels. This should be changed to use printf, a more general routine, for consistency.

Within the limits of these constraints, bar_graph works as advertised and may be sufficiently general to be of some use. It may also serve as a model for a new bar graph-making routine.

Map. The map display as currently implemented is the most specific of the three data display functions. It is so geared to one specific system as to be of virtually no general use. However, portions of the associated code could easily be modified to be generally useful.

The basic map functions include:

1. drawing the outline of a map,
2. shading the areas of the map according to the value associated with each area, and
3. displaying one shaded box of the map legend.

The code to perform these functions is driven by external data structures describing the map, and is fairly general. However this code is currently imbedded in other, more specific, procedures. The data structures themselves must be built on UNIX and loaded into the IT system when it

is built.

The map routines as they stand have two design shortcomings:

1. the names of the external data structures are assumed to be fixed, so that only one map per IT system is allowed, and
2. the entire data structure must be resident in memory.

PART II:

BUILDING AND EXECUTING IT SYSTEMS

OVERVIEW

Part II of this manual is concerned with the mechanics of building an IT application system. It is assumed that the reader has decided upon the application, and has written the code needed by the application process. From this starting point, the sections in this part discuss:

1. how to build and load an IT system, and
2. tips on how to debug a system that doesn't work.

Some discussions are very different for LSI-11 and for Level 6 terminals. In these cases, the two terminals are discussed separately.

Introduction

This section contains step-by-step instructions on how to construct an IT system and how to load it into the IT. The procedure and required tools to build and load systems are very different for the LSI-11 terminal and for the Level 6 IT. In order to keep the descriptions of the two procedures as clear as possible, separate discussions are presented: first for the LSI-11 and then for the Level 6. The reader needs only to read the subsection of immediate concern.

Within each subsection, the assumptions which have been made as to the availability of support tools are stated. Following these is an outline for creating an executable IT system and one for loading it.

LSI-11 IT

Assumptions. It is assumed that the reader has available as reference guides a UNIX Programmer's Manual [11] and the LSI-11 Processor Handbook [7].

Necessary hardware: It is assumed that the following hardware is available:

- . a working UNIX system,
- . operational hardware for the LSI-11 IT,
- . two floppy disks which are provided with the terminal, one with the Terminal system and one with the Phonenumber Loader system,
- . some way to transfer information between UNIX and the IT.

This could be a hardwired line or a dialed-in phone line

with a modem connecting the IT to UNIX, or the facility to write floppy disks on UNIX in the IT format.

Necessary software: It is assumed that the following software steps have been taken:

- . the reader can log on to UNIX, and has access to the necessary routines,
- . all of the IT system routines have been compiled and archived in the proper order into /lib/libI.a on the UNIX system,
- . code for user-supplied application routines has been written,
- . the interrupt vectors have been assembled and the object for this is in the file low.o in the same directory as the application routines, and
- . the UNIX program "boot" either has been made generally available or is in the same directory as low.o and the application routines.

If any of these have not been taken, the reader will need to contact a more experienced IT programmer so they can be done. Alternatively, the user can read the sections Building an IT System and Loading a Completed System onto the IT, and so become such a programmer.

Reader's knowledge: The UNIX commands chdir, cc, sh, ld, nm, strip, mv, msg, and stty are used in this outline. It is assumed that the reader is familiar with these commands or will check [11] for an explanation of them. Also the reader must be able to initially logon to UNIX.

Building an Executable IT System. Following are the steps to take in creating an IT system. Some of the steps note reference sources to check if problems are encountered.

1. Set up the IT as a standard terminal. This requires several steps:
 - . Turn on the power to the IT if it is off.
 - . Mount the floppy disk labeled "Terminal" on drive 0 of the IT. To properly orient the floppy, during insertion the user should hold the disk by the edge opposite the oval hole with the label side up.
 - . Push the "Start" button on the IT and the "IPL" button on the disk to automatically load the terminal simulator program.
 - . After it is loaded, the terminal system will hold a brief dialogue with the user. When this is complete, the IT will be set up as a standard terminal.

2. Logon to UNIX [11].

3. If necessary, change the working directory to the one which contains the application routines [11], e.g.

chdir IT

4. If the user-supplied application routines have been changed since the last time they were compiled (or if they have never been compiled) they should be compiled now. For example:

cc -c main.c subl.c sub2.c [11]

Note that the "-c" option to cc MUST be present. (If any compilation errors occur here, the affected routine(s) must be fixed and re-compiled before proceeding to the next step.)

5. Load the user-supplied application programs together with the system routines using the ld command [11]. For example:

ld -X low.o main.o subl.o sub2.o -lI -lc -la

Errors here are usually in the form of undefined external

references. This could mean that a subroutine or external variable name was misspelled in some routine, or that the files in a library are out of order. These errors must be fixed, the offending routines recompiled and the system reloaded before proceeding.

6. It is useful, but not necessary, to print a list of external symbols at this point for later debugging use. To do this, type:

```
nm -ng a.out    pr -4 -wl32    lpr                                [11]
```

This will cause the namelist to be printed on the UNIX line printer.

7. Now, it is also useful to strip the symbol table from the executable file [11]. This will not effect the execution of the system, but it will reduce the size of the file, thus reducing the disk space requirements. Type:

```
strip a.out
```

8. Finally, it is a good idea to rename the executable system file, called "a.out", which has just been produced. If this is not done, subsequent system constructions, assemblies, or compilations will destroy this system file. Type:

```
mv a.out IT_system                                [11]
```

"IT_system" in this example is the new name for the system file. It is not necessary to use this name. The user can use any valid UNIX file name which is convenient.

The system file has now been created. At this point it is possible to log off, if desired, or to proceed to Loading the System onto the IT.

Appendix F contains a listing of a file, `RUN_ME_mksys`, which can be executed to automatically perform steps 5 through 8 of the previous sequence. If this set of commands is put into a Unix file, it can then be executed by the `sh` command. The assumptions made by `RUN_ME_mksys` are included in its description in Appendix F.

Loading the System onto the IT. There are three ways to load an IT system into the terminal:

1. using the communication line between UNIX and the IT,
2. using the bootstrapping facility on the IT floppy disk, and
3. using the IT utility system SOTS.

Each of these techniques are described below.

Using communication line: Following are the steps to take for loading a system into the IT by using the communication line between the IT and UNIX.

1. If not still logged in to UNIX, re-do the first three steps of the last session, i.e., set up the IT as a terminal, get logged in and change the working directory to the one with the new IT system.
2. Disallow messages from other users by typing:

```
mesg n
```

 [11]
3. Force UNIX to not echo characters input to it by typing:

```
stty -echo
```

 [11]
4. The IT bootload program must be loaded at this time. To do this, mount the floppy disk labeled "Phoneline Loader" on drive 0. Push the "Start" and "IPL" buttons to automatically load and invoke the bootload program.
5. The bootload will use the keyboard display to ask if the

system file should be written to disk. To execute the system type an "n" followed by a carriage return. (New system disks can be created by copying the system file to disk. To do this, the user should remove the "Phoneline Loader" disk and mount a new blank disk, and then type a "y" instead of "n" to bootload. The bootload program will then write the system file to disk instead of executing it after it is loaded.) The bootload will ask for the name of the file to load. When it does, type the UNIX name of the system file ("IT_system" in this example), followed by a carriage return.

6. The IT bootloader will then use the UNIX "boot" command to pull the IT system in over the line. The IT system will be automatically started when transmission is completed (unless it is being written to disk). If the UNIX end of this procedure ("boot") can't find the specified file (usually due to a misspelling on the part of the user), then the bootloader will ask again for the name of the file. Check the spelling of the name and re-type it.
7. When the bootloader has successfully loaded the IT system for execution, it will logoff from UNIX and start the new system program. If it has copied the system to disk, the bootloader will not log off, but will loop and ask the user for the name of the next file to load.

Using floppy disk bootstrap: It is possible to use UNIX facilities to put a system directly on a floppy disk in the IT bootstrap format. This technique is generally somewhat simpler and faster than the previous one. The steps for creating a new bootstrap disk are described below.

1. Logon to UNIX, either through the IT as described above or using a standard terminal.
2. Mount the floppy disk to be used on any of UNIX's floppy disk drives. In general, any previous contents of the floppy disk will be lost in the process of writing the bootstrap program. The user should consider this when selecting a floppy disk to use.
3. Invoke the utility to write the system in bootstrap format, by typing:

`put_one_one`

4. `Put_one_one` will hold a short dialog with the user to determine what needs to be done. After each response, the user should type a carriage return. In particular, `put_one_one` will ask:
 - . the number of the drive which contains the floppy,
 - . whether the floppy disk should be initialized, and
 - . the name of the file containing the system to be put onto the disk.

To specify whether the floppy should be initialized, the user should type either a 'y' for yes, or an 'n' for no. If the disk has never been used before, it must be initialized. If the disk has been used it is not necessary to reinitialize it, but it will not hurt to do so.

5. When `put_one_one` has written the system to disk, it will ask again for the name of the drive to use. Enter a control D (EOT, octal 4) to return to command level.
6. Logoff of UNIX.
7. Remove the floppy disk and mount it on drive 0 of the IT.
8. Push the "Start" and "IPL" buttons on the IT to automatically

invoke the system.

Put_one_one can be used to create more than one bootstrap disk per invocation. To do this, the user merely continues answering the questions asked by put_one_one, taking care to ensure that the created disks are replaced by new floppy disks before reusing any drive. When all the system disks desired have been written, the user can return to command level by typing a control D (EOT, octal 4).

Using SOTS: The LSI-11 IT is provided with a utility system which can load the terminal with a program stored as a file in IT format on a floppy disk. This utility program is called SOTS and resides in the util subdirectory of the main it directory.

To load a system using SOTS, take the following steps.

1. Logon to UNIX.
2. Mount a floppy disk on drive 0 on UNIX.
3. Put SOTS on the disk in bootstrap format as described above.
If the disk already contains SOTS as the bootstrap, this step can be omitted.
4. Write the new IT system on the disk using the utility put.

Using the above example name, this would be done by:

```
put IT_system <IT_system
```

5. Logoff of UNIX.
6. Mount the floppy disk with SOTS and the new system on drive 0 of the IT.
7. Press the "Start" and "IPL" buttons on the IT to load SOTS.
8. When SOTS is invoked enter the name of the new system, IT_system in this example, followed by a carriage return.

Typing errors made when entering the name can be corrected by:

- . typing either backspace (BS, octal 10) or delete (DEL, octal 177) to delete the previous character, or
- . typing a cancel (CAN, octal 30) or control X (octal 30) to delete the entire line.

When the name is entered, SOTS will automatically invoke the new system. Some systems may be too large to load using SOTS. In these cases, SOTS will print a message saying that it is unable to load the system because it is too large. Such systems must be loaded using one of the two methods described above.

Level 6 IT

Assumptions

Necessary hardware: It is assumed that the following hardware is available:

- . a working UNIX system, and
- . operational hardware for the Level 6 IT.

Necessary software: It is assumed that the following software steps have been taken:

- . the reader can log on to UNIX, and has access to the necessary routines,
- . all of the IT system routines have been compiled, transferred to the Level 6, and assembled with the resulting object code being located in a file called LIB,
- . code for user-supplied application routines has been written,
- . a copy of the UNIX to Level 6 cross compiler, l6, is available on UNIX, and
- . the UNIX utility program tol6 has been made generally

available.

If any of these steps have not been taken, the reader will need to contact a more experienced IT programmer so that they can be done. Alternatively, the user can read the sections Building an IT System and Loading a Completed System onto the IT, and so become such a programmer.

Reader's knowledge: The UNIX command choir and the utilities 16 and tol6 are used in this outline. It is assumed that the reader is familiar with these commands. Also the Level 6 BES commands DEC2L6, ASM, LINKER, and AT are used. It is assumed that the reader is familiar with these commands or will go to [9] or Building a Level 6 System for explanations of them. Additionally, the user must be able to logon to UNIX.

Building an Executable IT System. Following are the steps to take when creating an IT system. Some of the steps note reference sources to check if problems are encountered.

1. Using a standard terminal, logon to UNIX.
2. If necessary change the working directory to the one which contains the application routines [11], e.g.

chdir IT

3. All user-application routines which have changed since they were last compiled should be re-compiled at this time using the UNIX to Level 6 cross-compiler 16. This compiler is used in approximately the same manner as the cc compiler. For example:

16 main.c sub1.c sub2.c

(Note that 16 does not require a -c parameter.) If any

compilation errors occur here, the affected routine(s) must be fixed and re-compiled before proceeding to the next step.

4. Transfer the files created by the compilation to the Level 6. Each file must be transferred individually. Steps 5 through 8 describe the transfer procedure for one file.
5. Mount a floppy disk on any drive on UNIX.
6. Write the output file to a floppy disk using the utility `tol6`. This utility will hold a dialogue with the user to determine exactly what needs to be done. An example of this dialogue which will put the assembly file corresponding to `main.c` above onto the disk which is mounted on drive 1 is:

Drive number (0 or 1): 1

Do you want to initialize this disk? (y or n): n

Name of file: main.a

(Remote file opened)

Drive number (0 or 1): [EOT]

7. Remove the floppy from UNIX, and mount it on the Level 6.
8. Read the file into the Level 6 file system using the utility `DEC2L6`. `DEC2L6` assumes that logical file number 1 is attached to the disk which was written by `tol6`, and that logical file 3 is attached to the user's library file. If `xxxx` is the disk drive with the disk created by `tol6`, `ULIB` is the user's library which is mounted on drive `yyyy`, and `file.a` is the name of the assembly-language file being

copied, then enter the BES commands:

```
AT 1 DSKxxxx DUMMY
```

```
AT 3 DSKyyyy ULIB
```

```
DEC2L6 FILE.A
```

(If a number of routines are being copied into the same library file, the first two lines need only be typed the first time.)

9. After all files have been transferred to the Level 6, log off of UNIX.

10. Assemble the transferred files. First enter the BES command

```
AT 1 = 3
```

to allow the assembler to read the files that were just put into ULIB. Then, for each file name.A that was copied, enter the command:

```
ASM name NL
```

(Note that the ".A" suffix is left off of the file name.)

This will produce an object file with the suffix .0 for each source file.

11. (Optional) To save space in ULIB, it might be useful to delete the assembler source files from the library. To do this, assuming ULIBD is the name of the disk on drive yyyy which contains ULIB, type:

```
AT 10 DSKyyyy ULIBD
```

```
UTILL1
```

```
DL ULIB:file.A 10
```

(Repeat the above line for each file to be deleted.)

```
QT
```

DT 10

12. Load the files to create the IT system. The file ITSYS.S in LIB contains the linker commands required to build a system. To use the linker with these commands, type:

AT 4 DSKzzzz LIB

AT 5 =4

AT 2 LPT0580 Note: this step is used only when there is a line printer.

LINKER ITSYS.S

where zzzz is the disk drive containing LIB. This will link all of the object files in ULIB together with the necessary object files from LIB. The resulting IT system will be contained in the file ITSYS in ULIB.

The system has now been created. At this point it is possible to log off of the Level 6, if desired, or to proceed to Loading the System Onto the IT.

Loading the System Onto the IT

To load the system onto the IT, it is necessary to tell the BES command processor which file contains the system and what is the name of the system. If yyyy is the disk containing ULIB, then the necessary commands to load the system are:

AT 0 DSKyyyy ULIB

ITSYS

BUILDING AN IT SYSTEM

Introduction

This section describes, in greater detail than the Cookbook section, how to build a system for either an LSI-11 IT or a Level 6 IT. It covers the major phases of system construction, explaining what needs to be done and why. Again, the methods for creating systems for the LSI-11 terminal are very different from those for the Level 6 IT, and are presented in different subsections.

Building an LSI-11 IT System

System creation is done in four major steps:

1. The source code for the system is created in several files.
2. The source files are compiled or assembled, yielding loadable object files.
3. Object files for system routines and commonly used application routines are collected together into one or more libraries.
(This step is optional.)
4. The various object files are loaded together using the UNIX `ld` command to produce the single, executable system file.

All procedures (application and system) and data structures needed by the application system must be loaded together in one executable file.

Note that all UNIX commands referenced in this section are explained in the UNIX Programmer's Manual [11].

Before discussing each of the steps required to produce a system, there is a brief discussion of UNIX naming conventions. Following this, each of the major steps is discussed in turn. Finally there is a brief section of miscellaneous notes.

Naming Conventions. The files containing IT code follow a standard set of naming conventions. By understanding and using these conventions, programmers can tell a great deal about the contents of a file simply by its name and can help avoid inadvertently overwriting files.

Source code file names end with the suffix ".c" or ".s". The ".c" suffix is required by the C compiler for all C language source files. The ".s" is used to designate assembler code source files.

Loadable object files have the suffix ".o". This naming format is followed automatically by the C compiler: compiling name.c will produce name.o. However the assembler always leaves its output in the file a.out. In order to avoid overwriting the assembled file, users should explicitly rename the file to the corresponding ".o" name.

For simplicity, executable files have no suffix.

There is a fourth type of file used by the IT code. This is called an "include file" and is usually, but not always, recognizable by its ".incl" suffix. Include files are widely used to hold the definitions of system values or structures. These files may be referenced by other files by means of the C "include" construct. This causes the entire contents of the included file to be inserted into the source file at the point it is referenced.

The function of each of these types of files in the process of building a system is discussed below.

Creating Source Files. Any of the standard UNIX editors can be used to create source files. Source files may also be loaded from a tape or other machine-readable medium. The include files referenced by IT source files must also be created, as above, before the source files can

be processed. Before compiling the system it may be necessary to change some of the values defined in these files to reflect the exact requirements of the application system.

Creating Loadable Object Files. Once each source file (and all its include files) is available, a loadable object file is created from it. This is done using the UNIX command "cc" for C language programs or "as" for assembly language source. It is necessary to use the "-c" option with cc or the "-" option with as to insure that loadable code for the routine is produced. Without these options, the cc command will attempt to create an executable file using the IT routine and the standard UNIX routines. This will produce erroneous results. Also, after each assembly, the resulting a.out file must be renamed to avoid being overwritten by the next assembly or compile. For example, after assembling the file exam.s, the user should do a "mv a.out exam.o" to preserve the assembled file.

It should be noted that unless this is the first time the system is being built, it is not necessary to re-compile all of the routines. It is only necessary to compile those routines which have changed since the last compilation.

Building Libraries. It would be possible at this point to create an executable IT system file from the loadable code files (distinguished by their .o suffix) produced in the previous step. However, this would be very tedious since a large number of files would have to be specifically named. It is extremely useful to first build one or more libraries containing system routines and any application routines which will not change frequently. Routines which are still being debugged and are therefore changing frequently will generally be left out of the libraries. The use of libraries will make it unnecessary to list all of

the routines in the system when they are loaded. It also has the advantage that library routines will automatically be included only if they are referenced by some other routine, thus minimizing the size of the system.

The UNIX `ar` (archive) command is used to build a library. A library is an archive whose members are loadable object files. A library of system routines is useful for making systems, as the user can pass the library to `ld` and all the system routines in the library used by the current application will be extracted. This manual assumes that such a library exists in the file `/lib/libI.a`.

Because of the way libraries are treated by the loader, the order of files within a library is significant. To understand why, and to be able to order entries within libraries properly, it is necessary to know something of how the loader operates.

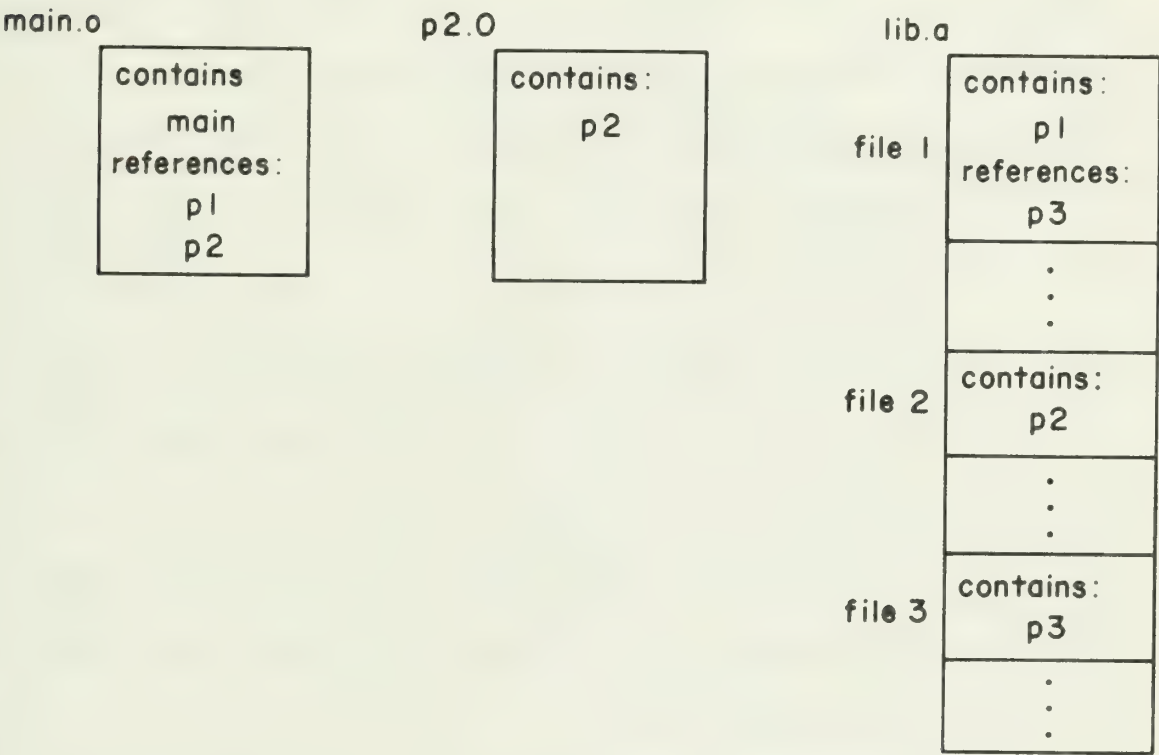
As the loader loads each file, it will

- . resolve any external references in the new file that are supplied by already loaded files,
- . use the external references in the new file to resolve any references left over from previous files, and
- . add the still unresolved references from the new file to the list of undefined symbol names.

The arguments to the loader will be scanned in the order they are listed on the command line. If an argument is an object file, it will be included when it is encountered. If the argument is a library, then the loader will look at each file in the library in turn. For each file in the library, the loader will check to see if that file can resolve any previously undefined names. If it can, the file is included in the loaded system. Otherwise, it is not. As a result of this action, the order of files in a library is important. For example, file `printf.o` contains the procedure `printf`. `Printf` references the procedure

put ascii which is contained in put ascii.o. In order for the loader to pick both printf and put ascii from the library, printf.o must precede put ascii.o in the library. Figure 15 is an example of a load using a library.

Typical problems which occur when the files in a library are not in the correct order include:



When these files are loaded together by the command
ld main.o p2.o lib.a
the resulting system will include the contents of main.o, p2.o file 1, and file 3. File 2 will not be used since p2.o will resolve the reference to p2.

Ld using a library

Figure 15

- . an undefined global variable,
- . a missing procedure, or
- . an incorrect version of some procedure or global variable being included, possibly causing unexpected results when the system is executed.

Loading the Executable System. The last step required for producing the executable system file is to load the pieces together. The order in which parameters are specified to ld is significant since routines will occur physically in the loaded system in the order in which they are encountered, and due to the use of libraries. The general order of the parameters should be:

1. low.o. Because of the way interrupts are handled on the LSI-11, the first 400 (octal) bytes of memory are reserved for interrupt vectors. In order for these to be properly initialized, the file containing the interrupt vectors must be included first. This will usually be low.o.
2. System specific application routines. Any user-supplied application routines specific to this system should generally be included next.
3. Application libraries. These are general application routines (or libraries containing them) that might be included in the system.
4. The IT system library /lib/libI.a. This may be referenced by the shorthand notation -II.
5. The standard UNIX libraries. These are specified by the -lc and -la options, and make it possible to use standard UNIX routines where desired.

The result of this load will be an executable system named a.out. This should be renamed to avoid being inadvertently overwritten.

An example of an ld command to build an IT system, which assumes that all of the application object code is in one subdirectory off of the main IT directory, would be:

```
ld ../kernel/low.o *.o -lI -lc -la
```

This will produce an output file which contains:

1. the version of low.o in the kernel subdirectory off of the main IT directory,
2. all of the object files in the application subdirectory, and
3. support facilities from the IT library /lib/libI.a, and from the UNIX system libraries /lib/libc.a and /lib/liba.a as needed.

When the load has completed the system should be renamed by doing a

```
mv a.out X
```

where X can be any UNIX file name desired. After the system has been renamed, other systems can be loaded without destroying this system.

For later debugging purposes, the user may want a namelist, described in Debugging IT Systems, of the new system. This can be generated by the UNIX command nm. After the namelist is generated, the symbol table of the a.out file should be removed by the strip command. Strip will not affect the ability of the file to be executed, but it will reduce the size of the file as stored on disk.

Some Random Notes on Building Systems. The system initialization routine startup will create one user process called "main" in addition to a process for each device in the system. The name of the

user process can be changed by changing the values in PROCTAB.c before building the system.

If the user needs to replace a system routine, this can be done by explicitly specifying the user file containing the procedure, with the same name, in the `ld` command. The file must be specified before the library which contains the system routine to be replaced.

Building a Level 6 IT System

System creation is done in five major steps:

1. The source code for the system is created in several files on UNIX.
2. The source code is compiled on UNIX, producing files containing Level 6 assembler code.
3. The assembler files are transferred to the Level 6.
4. Files are assembled on the Level 6.
5. The various object modules are linked together to produce a single, executable system.

All procedures (application and system) and data structures needed by the application system must be loaded together in one executable file. Note that all UNIX commands referenced in this section are explained in the UNIX Programmer's Manual [11]. BES commands are explained in Program Development Tools [9].

Before discussing each of the steps required to produce a system, there is a brief discussion of UNIX naming conventions. Following this, each of the major steps is discussed in turn. Finally, there is a brief section of miscellaneous notes.

Naming Conventions. The files containing IT code follow a standard set of conventions. By understanding and using these

conventions, programmers can tell a great deal about the contents of a file simply by its name and can help avoid inadvertently overwriting files.

Source code file names end with the suffix ".c" or ".a". The ".c" suffix is required by the 16 C compiler for all C language source files. The ".a" is used to designate assembler code source files. The ".a" is used to designate assembler code source files. Output files from the 16 compiler will automatically contain the ".a" suffix. This suffix is also required by the Level 6 assembler.

Loadable object files have the suffix ".o". This naming convention is automatically followed by the Level 6 assembler: assembling name.a will produce name.o.

For simplicity, executable files have no suffix.

The Level 6 restricts file names to being at most six characters plus a suffix. As a result, the names of files on the Level 6 are generally truncated versions of the UNIX names. For example, compiling the file "long_name.c" on UNIX will produce an output file named "long_name.a". When this is transferred to the Level 6, the name will have to be shortened. In order to retain the ".a" suffix, the user must truncate the first part of the name so that "long_name.a" on UNIX becomes "long_n.a" on the Level 6. When this file is assembled, its resulting object file will automatically be named "long_n.o".

There is a fourth type of file used by the IT code. This is called an "include file" and is usually, but not always, recognizable by its ".incl" suffix. Include files are widely used to hold the definitions of system values or structures. These files may be referenced by other files by means of the C "include" construct. This causes the entire contents of the included file to be inserted into the source file at the point it is referenced.

Creating Source Files. This process is the same for Level 6 systems as for the LSI-11 systems. See the subsection Creating Source Files in the previous discussion.

Creating Assembly Language Files. Once each source file (and all its include files) is available, a Level 6 assembly file is created from it. This is done using the UNIX to Level 6 cross-compiler 16. It should be noted that unless this is the first time the system is being built, it is not necessary to recompile all the IT routines. It is only necessary to compile those routines which have changed since the last compilation.

Transferring Files to the Level 6. Files are transferred to the Level 6 by writing them on floppy disks using the UNIX utility `tol6`, and then reading them using the Level 6 utility `DEC2L6`. Only one file can be put on a floppy at a time, so if several files are to be copied it is necessary to use several floppies or to read each file off of the floppy before the next is put on.

Files are written to floppies using the utility `tol6`. `Tol6` removes any previous file on the disk and then allows the user to specify the name of a file to write. When it is invoked, `tol6` engages in a dialogue with the user, asking the following questions:

1. Disk to put file onto (0 to 1):
2. Do you want to initialize this disk first (y or n):
3. Name to initialize disk with:
4. File name:

The possible responses to the first two questions are noted in the questions themselves. (A disk only needs to be initialized the first time it is used for an IT file.) The third question will be asked

only if the user responded "y" to the second question, and should be answered with any ASCII string of characters indicating the desired name of the disk. The last question should be answered with the name of the UNIX file containing the IT file to be transferred to the Level 6.

The file written onto the floppy is organized as a linked list of disk sectors. The file starts on track 0, sector 1. The last two bytes of each sector contain the track and sector number of the next piece of the file. The end of the file is indicated by a sector number of zero.

Once a file has been written to a floppy disk, it must be copied into a BES partitioned file using the utility DEC2L6. (See the BES utilities manual [12] for instructions on how to create and format a partitioned file.) DEC2L6 expects logical file number 1 to be attached to the floppy written by tol6 and logical file number 3 to be attached to the partitioned file which will receive the copied file. (In standard BES systems, the id's of the two floppy disk drives are, from left to right, 0400 and 0480.) If xxxx is the id of the drive containing the disk written by tol6 and yyyy is the id of the drive with the user's library (partitioned file) called ULIB, then use the two attach commands

```
AT 1 DSKxxxx DUMMY
```

```
AT 3 DSKyyyy ULIB
```

Then, to copy each file, type:

```
DEC2L6 FILE.A
```

where FILE.A is the name that will be given the file in ULIB.

Assembling the Files. Once the assembly-language files have been copied to the Level 6, they must be assembled. The assembler reads its input from logical file number 1 and writes its output to logical file number 3. It is possible to write the object file back into the same partitioned file that contains the assembler source by attaching both logical file numbers 1 and 3 to the same file. If an assembly-language listing of the program is needed, logical file number 2 must be attached to the line printer. To assemble the file FILE.A in ULIB on disk yyyy with no listing and put the object into ULIB, use the commands:

```
AT 1 DSKyyyy ULIB
```

```
AT 3 =1    (This means that logical file 3 is the same as 1)
```

```
ASM FILE NL
```

Note that the ".A" suffix is left off of FILE in the ASM command.

If a listing is desired, use the commands:

```
AT 2 LPTpppp
```

```
AT 1 DSKyyyy ULIB
```

```
AT 3 =1
```

```
ASM FILE
```

where pppp is the id of the line printer. (This is 0580 in standard BES systems.)

The assignments made in the AT commands remain until they are explicitly changed (or until the system is re-booted). Specifically, the ASM command does not destroy them. Therefore, if several files are to be assembled from the same library it is only necessary to enter the AT commands the first time.

Linking the Executable System. Once all of the necessary files have been assembled, the object files must be linked together to form an executable system. The BES linker requires a number of attach commands. The logical file numbers that must be attached are:

1. This should be attached to the user's library.
2. This is where the linker's list of files included and the map will be printed. It can be attached to either the system console or to the line printer.
3. This should be attached to the partitioned file in which the executable file should be put. It can be the same library that contains the user's object files.
4. This defines the device or file which contains the linker's command input. To use the provided command stream ITSYS.S described below, this should be attached to LIB.
5. Attach logical file number 5 to LIB, the library of IT system software.

If the IT library LIB is on drive zzzz and the executable program is to be put into ULIB, use the set of attaches

AT 1 DSKyyyy ULIB

AT 2 LPTpppp

AT 3 =1

AT 5 DSKzzzz LIB

AT 4 =5

The IT library LIB contains a file ITSYS.S containing the linker control commands required to load a system. The commands included and what they do are as follows:

NAME ITSYS

This command specifies the name of the executable file to be produced. The executable file produced by ITSYS.S will always be ITSYS.

HMA X'FD00'

This specifies that the linker can create a loadable file up to FD00 (hex) words long.

LINK LOW

This forces the file LOW.0 to be loaded first, thus ensuring that the low end of memory will be set up properly when the system is executed.

LINKA

This causes all object files (files with a ".0" suffix) in the file attached to logical file number 1 to be included. Therefore, all object files in ULIB will be included, even if they are not called by any other routine.

LINK CS8X16,CS6X10

This loads the two character sets.

LINKN END

This defines the external variable END used by startup to locate the highest address used by the IT system.

MAP

This forces the files mentioned above plus all necessary files in LIB to be actually read in. A map of procedures and external variables is printed along with the address in memory of each. Also printed is a list of procedures and any undefined external names that have been referenced.

If any names are undefined, there is an error. Check for a spelling error or for a missing object file.

QT

This causes the linker to write out the executable system file and quit.

To run the linker using the above set of commands in ITSYS.S, do the indicated attaches and enter the command

```
LINKER ITSYS.S
```

There is one important caveat when using the linker. As the linker finds and includes each object file needed, it prints a line saying that that file is included. If it cannot find a file (named, for instance, FILE.O) that is needed it prints the message

```
FILE.O NT FND
```

Unfortunately, for files in the user's library, this message is sometimes printed in error. If this message appears in the linker listing, check the list of undefined names. If there are no unexpected undefined names, then the error message can be safely ignored. If, however, the name is undefined, then the file was really not found. This is usually due to a missing file or to a spelling error on the part of the user.

Some Notes on Building Systems. The system initialization routine startup will create one user process called "main" in addition to a process for each device in the system. The name of the user process can be changed by changing the values in PROCTAB.c before building the system.

If it is necessary to replace some system routine in a given IT system with one supplied by the user, this can be easily

accomplished by including the new copy of the routine in the user's library. The LINKA command to the linker will ensure that that routine gets included, and it will override the system routine with the same name.

LOADING A COMPLETED SYSTEM ONTO THE IT

Introduction

Once an executable IT system file has been created using the procedures described in Building an IT System, it is necessary to bootload the file onto the IT hardware. This chapter explains this process for the LSI-11 IT and then for the Level 6 terminal. For a step-by-step description of what needs to be done, see the section Cookbook.

Loading the LSI-11 IT.

As mentioned in the section Cookbook, there are three methods of loading a system into an LSI-11 IT. One method involves dumping the system to the IT from UNIX, using the connecting communication line. The second method involves using UNIX to write the system on a floppy disk in IT bootstrap format and then bootstrapping from the disk. The third method involves using the IT utility program SOTS to load the system. Each of these methods is described below, followed by a discussion of their relative merits.

Using Communication Line. The bootloading process requires two cooperating programs, one on the IT and one on UNIX. The program boot, running on UNIX, sends the IT system over an 8-bit data path. The bootloader program running on the IT reads this data into successive locations of the IT memory, starting at location 0. When the entire system has been read in, the bootloader logs out from UNIX and branches to location 0, thus starting up the IT system.

Boot first sends a two-byte header which is the length of the system file in words. Then it sends the file itself one byte at a time.

(Note that on some UNIX systems it is not possible to transmit 8-bit binary data. In this case boot will break each 8-bit byte into two 4-bit pieces and send each piece as a separate byte. The bootloader on the IT must then reassemble the two pieces back into one 8-bit byte.)

Doing the bootload: The IT should be set up as a standard terminal by loading the terminal simulator from the disk labeled Terminal. In this mode, the IT should be used to connect to UNIX and to enter a directory where the system file and boot are both accessible. It is useful to have made boot a generally available command on UNIX.

Once this has been accomplished, the bootloader program should be loaded into the IT. This program resides on the disk labeled "Phoneline Loader" and will overwrite the terminal simulator when loaded. The bootloader can perform two different types of loads. In one case when the IT system is loaded, bootload will log off of UNIX and branch to the location 0 to start executing the system. In the other case, the loaded system will be written to floppy disk and bootload will loop to the point of asking for the name of the next file to copy. (In the second case, bootload will not log off of UNIX.)

When the bootloader is invoked it will ask, using the keyboard display, whether the file should be written to disk. The user should respond by typing a "y" if it is, or "n" if not, followed by a carriage return. The bootloader will then ask for the name of the file to be loaded. The user should respond by typing the name of the UNIX file. When this has been done, bootload will send the command "boot X" (where X is the name of the file to be loaded) over the line to UNIX, thus starting the file transfer.

Notes: It is assumed that the system file to be bootloaded is

in the same format as files created by the `ld` command.

It is a good safety precaution to disallow messages before starting up the bootloader. If someone writes to the terminal while a bootload is in progress, it can garble the system being loaded. This can be done by typing a `"mesg n"` command on UNIX. Unfortunately, it is not possible to disallow broadcast messages in this way.

Using Floppy Disk Bootloader. Under this method, UNIX is used to write the system onto a floppy disk in IT bootstrap format. After this has been done, the hardware bootstrap capability of the IT can be used to load the system into the terminal.

The IT system is written to floppy disk using the UNIX utility program `put_one_one`. (This utility can also be invoked by the name `pll`. The shorter name is provided for programmer ease.) `Put_one_one` removes any previous bootstrap program that was on the disk and then writes the new IT system onto the disk in the bootstrap format. After the disk has been written, `put_one_one` loops to allow the user to write another bootstrap disk. After the bootstrap system has been written, the disk can then be mounted on the IT and the disk's hardware bootstrap facility used to load the system.

When `put_one_one` is invoked, it will hold a dialogue with the user in order to determine exactly what needs to be done. The questions presented to the user are:

1. Disk to put startup program on (0 or 1):
2. Do you want to initialize this disk first (y or n):
3. Name to initialize disk with:
4. File name:

The possible responses to the first two questions are noted in the

questions themselves. The third question will be asked only if the user responded 'y' to the second question, and should be answered with any ASCII string of characters indicating the desired name of the disk. The last question should be answered by the name of UNIX file which contains the IT system to be written.

An IT bootstrap program on the floppy disk is organized as a linked list of disk sectors. The program starts in track 0, sector 5. The last two bytes of each sector contain the track and sector number of the sector containing the next piece of the program. The end of the program is indicated by a track and sector number of 0.

Using SOTS. SOTS is an IT system which can load other systems from floppy disk into the IT and start their execution. The loaded systems are stored on disk as standard IT files. The advantage of using SOTS to invoke programs is that several systems can be stored on a single floppy disk. This significantly reduces the number of floppies used to hold systems, as compared with the previous method which requires one disk per system.

To load an IT system using SOTS, merely type its name. The program will be loaded and execution started automatically. In addition, SOTS can also list the contents of a file. By typing

ls n

it is possible to get a list of the files mounted on drive n. To get a list of files on all of the drives, type ls with no parameters.

Comparison of Loading Methods. Each of the methods for loading an LSI-11 IT described above has some advantages and some disadvantages. Significant points regarding the method using the communication line include:

1. does not require that UNIX support floppy disks or that UNIX be able to write disks in the IT format,
2. is a slow process since loading is limited by the speed of the communication line,
3. there is no error detection in the loading process, and
4. unless the system is written to floppy disk when it is loaded, the entire procedure will have to be repeated in order to reload the system.

The first point is the major advantage to this method. Significant points regarding loading an LSI-11 IT from floppy disk include:

1. requires that UNIX have facilities to write floppy disks in IT bootstrap format,
2. is generally faster to write the system on UNIX and then bootstrap it on the IT,
3. UNIX facilities do error detection when writing the disk, so the chance of errors is reduced, and
4. this method automatically creates a disk for future use, eliminating the need to repeat the entire procedure later.

The first point is the major drawback to this method. If such facilities do exist on UNIX, then this will generally be the preferable method.

Loading the Level 6 IT

There are two methods for loading a system onto the Level 6 IT. The recommended method is to use the BES command processor. It is also possible to set up a BES system disk that will automatically bootstrap an IT system instead of the BES command processor. This is useful if there is no operator console on the IT.

Using BES Command Processor. Logical file number zero is used by the BES system to identify which file to load system from. Therefore, if a system names ITSYS in file ULIB on drive xxxx is to be loaded, use the BES commands:

AT 0 DSKxxx ULIB

ITSYS

Using BTGEN on a Disk. The BES program BTGEN can be used to identify which program will be bootloaded from a disk. To create a disk that will bootload an IT system, some setup must be done.

1. Create the partioned file PROGFILE on the disk using the BES utility package UTILL1. The use of UTILL1 is described in [12].
2. Copy the programs DSKLDLDR and TRPHND into PROGFILE from the file PROGFILE on the BES system disk. You will need to use UTILL3 to make the copies.
3. Use BTGEN to identify the name of the program to load when bootloading from the disk. BTGEN is described in detail in [12]. If ITSYS is the name of the program to be loaded, the parameters to use are:

N,N,FFFF,0,,ITSYS

4. Copy or link the system into ITSYS in PROGFILE. Linking the system is described in Building an IT System.

The disk is now ready. The operations needed to bootload the system are exactly the same as those used to bootload a BES system.

Comparision of Loading Methods. Each of the methods for loading a Level 6 IT described above has advantages and disadvantages. The significant points regarding using the BES command processor include:

1. It is relatively less convenient and somewhat slower. Several lines of input must be typed for each execution of an IT system.
2. Several IT systems can be stored on the same disk.

The second point is the major advantage to this method. By comparison, using BTGEN is:

1. More convenient, and
2. wasteful of floppy disk space, because only one program can be stored on each disk.

After an IT system has been successfully written, compiled, loaded, and dumped onto the IT, it may not execute properly due to system bugs of one sort or another. When this happens a great deal of information can be obtained by checking the values of various memory locations of the terminal.

This section includes tips and information useful in performing such post-mortem investigations on the IT system. It describes:

1. how to locate specific variables in memory,
2. commonly referenced system variables,
3. the organization of a process stack, and
4. an example of a post-mortem.

The discussion assumes that the reader is familiar with the use of the LSI-11 debugger, described in [7], and the use of the Level 6 console, described in [5].

Getting Started

Before the contents of memory locations can be meaningful, the programmer must know the correspondence between memory locations and the procedures, variables and data structures on the system. For LSI-11 terminals, the UNIX utility program nm can be used at system creation to produce a list of such correspondences. For Level 6 terminals, the list can be generated by a MAP command when the system is being LINKed. Each of these utilities are described below. A list of this sort is essential to understanding the contents of the IT memory.

Accessing Memory

UNIX nm. The nm command is used to produce a namelist for LSI-11 systems. Nm must be run after the entire system is loaded together and before it is stripped of symbolic information. The output

of nm is a printable file referred to as a namelist containing the memory address of variables in the system.

The exact variables included in the namelist will depend on the parameters passed to nm. Typically, it will include procedures and global (external) variables. The value associated with each variable is the address of the location referenced by the variable. For procedures, the value is the address of the first instruction of the procedure. For external variables, the value is the address where the variable is stored. For more complicated data structures the value is the address of the first byte of the space used by the variable. (Some structures written in assembler language may not follow this convention.) See [11] for a complete description of the nm command and its output.

Level 6 MAP. The MAP command can be used in the LINKER to produce a namelist for Level 6 IT systems. MAP will print the name of every file linked in the system, the started address of the file, and the name and address of all external variables defined in the file. The files in this list are sorted in order of increasing memory addresses. MAP also prints the starting address for the system, and the names of any unresolved external references. MAP is further explained in [9].

Useful Variables

Several system variables are particularly useful when debugging a system. These variables are noted here along with comments on how the variable is used, where in this manual the structure is diagrammed and other pertinent information.

ME. This holds the one word identifier of the process that was running when the system stopped. Since interrupt handlers do not affect the value of ME, the value of ME will not be correct when an interrupt is being serviced.

READY Q. This is a pointer to queue of processes ready to run. The ready queue is diagrammed in Figure 12. Note that the currently running process (ME) will not be on the ready queue.

FREE PTR. This is a pointer to a linked list of free queue elements. These queue elements are used for building the input queues for the various processes. If FREE PTR is 0, there are no more available queue elements. The free queue list is diagrammed in Figure 11.

PROCTAB. This is one of the most useful system variables when performing post-mortems. PROCTAB is a structure which contains pertinent information about all the processes created automatically at system initialization. Most of this information is supplied by the programmer when the system is created. However, as each of the processes is created its process identifier is stored in PROCTAB. Since the process identifier is the address of the base of the process stack, this variable will tell the programmer where to find the stack for each of these processes. This is essential to tracing the activities of the system before it died. PROCTAB is diagrammed in Figure 8.

DEV TAB. This is the device table for the I/O system. It contains information such as the address of the input queue for the process handling each device, the identifier of the owner of each device and the information in the last request block used for the device. This structure is diagrammed in Figure 13.

KBD Q, PP Q, TP Q, VIP Q, DSK Q. These are the queue heads of the input queues for the various device processes. Their structure is diagrammed in Figure 10. These queues can be checked to determine any commands written to each handler but not yet read by them.

Registers. Of course the value of the general registers, including the PC and the status word, are often useful. The meaning of the registers depends on the type of IT being investigated.

LSI-11 registers: The meaning of registers R2 through R4 will vary depending on the currently executing procedure. R0 and R1 are used for returning values from subroutines. R5 is used, by convention, to mark the base of the current stack frame. This convention is described further below. R6 and R7 follow the standard PDP-11 convention of being the stack pointer and program counter respectively.

Level 6 registers: C procedures on the Level 6 follow the Honeywell convention of using B5 for subroutine linkage. Base register B4 and general registers R6 and R7 are used for returning values from subroutines. The stack and environment linkage pointers are stored in B6 and B7 respectively. The use of these registers is further described below.

IT Process Stacks

Each process on the IT has its own stack space. The contents of these stacks can be very useful in debugging the system. The stack structure is described in this section. However before going into the details of the stack, a discussion of C conventions for stack usage on the LSI-11 and the Level 6 is presented.

C Conventions. C uses the stack for storing automatic variables, for passing parameters, for subroutine linkage, and occasionally for temporary storage space. On the LSI-11 C follows standard PDP-11 conventions, and uses register 6 (R6) as the top-of-stack pointer. On the Level 6, base register 6 (B6) is used as the stack pointer. Stacks grow from larger address locations to smaller addresses, referred to as "up". Additionally one register (R5 on the LSI-11 and B7 on the Level 6) is used as the pointer to the base of the current stack frame.

As each procedure is called, C sets up a stack frame for it and points R5/B7 at the base of the stack frame. Below this (at higher numbered memory addresses) are the stack frames for the calling procedures. The local stack frame is used for:

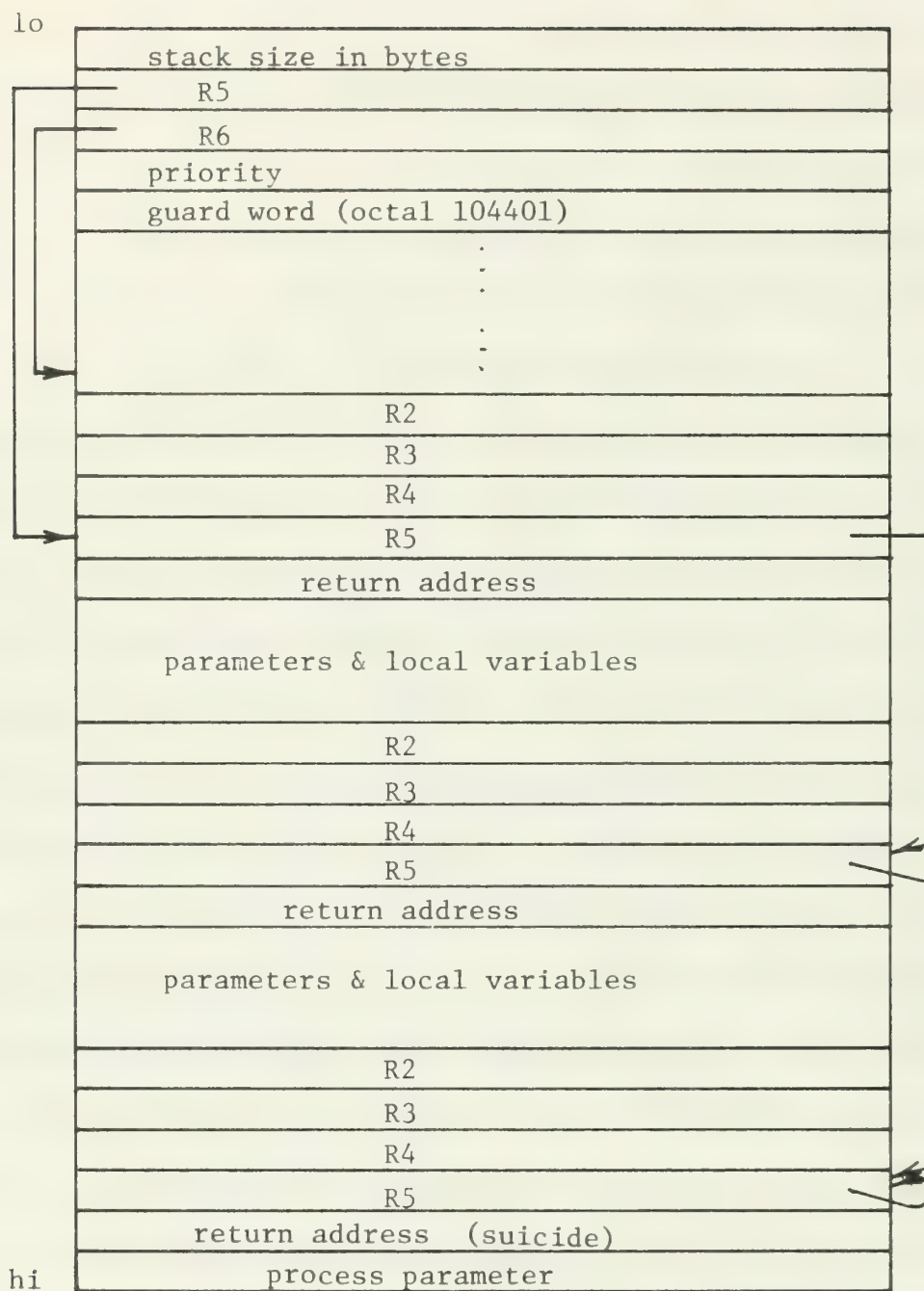
1. storing register values for the calling procedure,
2. local variables,
3. passing parameters and subroutine linkage, and
4. temporary storage.

Each of these functions is explained below:

Registers: Registers whose contents must be maintained across subroutine calls are stored immediately upon entrance to a C procedure. The details of saving these registers differ between the two systems.

In UNIX C, register storage is done by the subroutine csv. The saved registers are R5, R4, R3, and R2, in that order. Registers 2, 3, and 4 are used for register type variables in user procedures, and so need to be saved for consistency. R5 points to the base of the previous stack frame. It is saved so that the previous procedure environment can be restored upon return. After the registers are saved, R5 is updated to point to the base of the new stack frame, which is the location where the previous R5 is stored. Thus, R5 points to the latest link in a chain of subroutine environments. Figure 16 illustrates this linkage.

In Level 6 C, the two instructions which save the registers and update B6 and B7 are done in the procedure itself. All registers, except those which are used for returning values, are stored. Thus the thirteen registers B7, B6, B5, B3, B2, B1, I, R5, R4, R3, R2, R1 and M are put on the stack. They are stored contiguously such that B7 is stored at the lowest address and M is stored at the highest. By convention, B7 points to the base of the caller's stack frame. B7,



Subroutine environment linkage - LSI-11

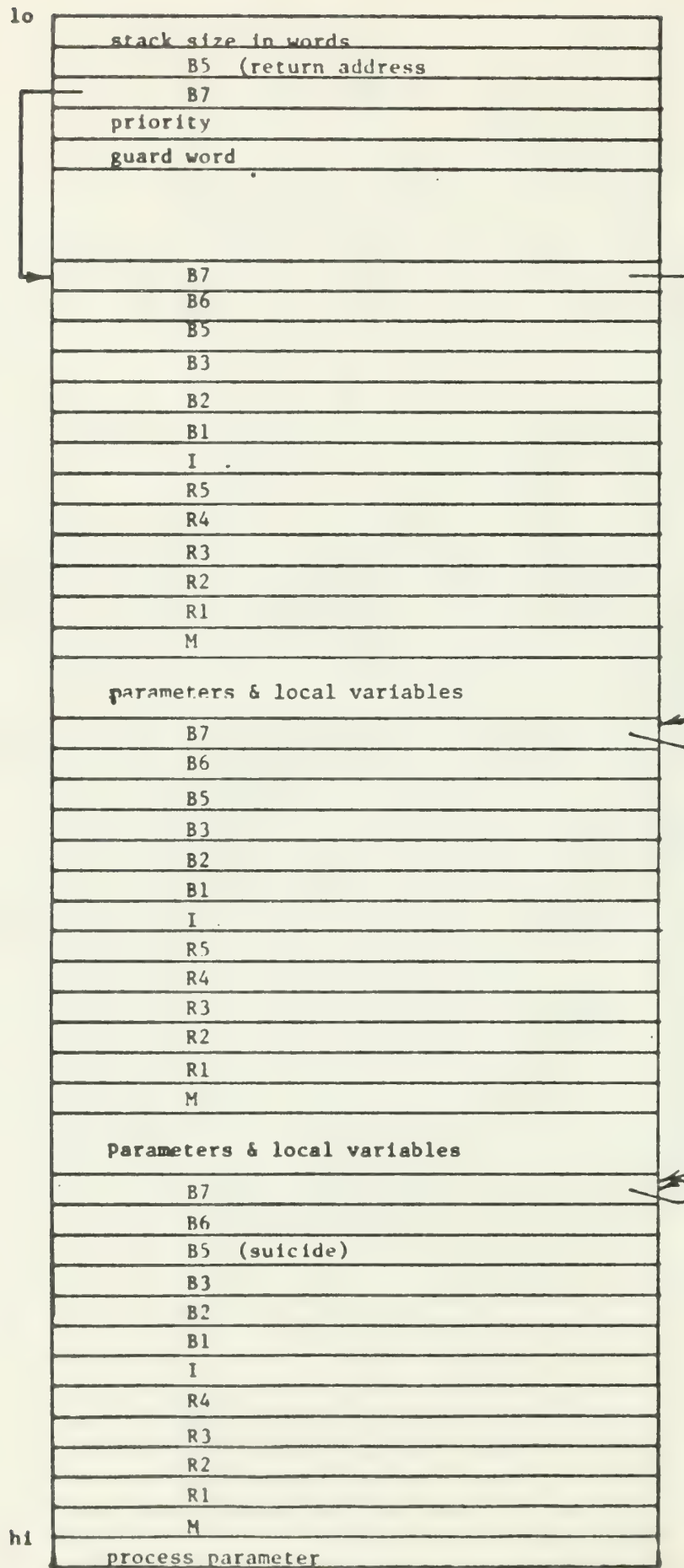
Figure 16

like R5 on the LSI-11, points to the latest in a chain of subroutine environments. Figure 17 illustrates this linkage for Level 6 IT's.

Local variables: On the stack immediately above the saved registers is the space used for the automatic local variables of the procedure. Space for each variable is pushed on the stack in the order in which the variables are declared in the C procedure. Thus if variable A is declared before variable B in the procedure, then space for A will be allocated on the stack before space for B; the memory location of A will have a larger address than that for B. Space on the stack for complex variables such as arrays or structures is arranged so that the lowest address is associated with the beginning of the variable. Figures 18 and 19 illustrate the correspondence between the declaration of variables and their position on the stack.

Not all internal variables in a procedure will use space on the stack. In particular variables declared static will not. Static variables have a fixed amount of space permanently allocated for them in the system. In UNIX C register variables use the general purpose registers R2, R3, and R4. The registers are allocated such that R4 is used by the first declared register variable, R3 by the second and R2 by the last. On the Level 6, no registers are used for programmer variables so all internal, non-static variables are stored on the stack.

Parameters and subroutines: To invoke a subroutine, C causes the parameters to the subroutine to be pushed on the stack in reverse order, so that the first parameter is on top, and then calls the subroutine. The subroutine will then save the general data register values and set up its stack frame just as the calling procedure had. Figures 20 and 21 illustrate the state of the stack after the subroutine has been called, but before it has saved any registers.

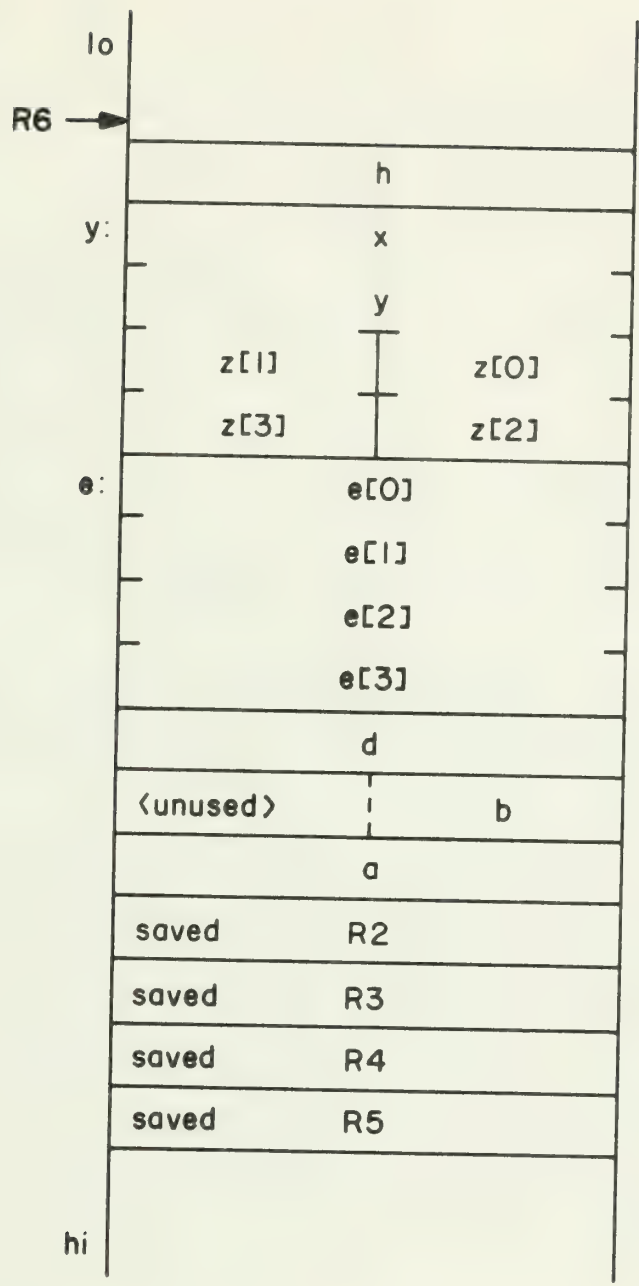


Subroutine environment linkage - Level 6

Figure 17

```
procedure ( )
{
int a;
char b;
register int *c;
int d;
int e [4];
static char f[100];
struct {
    int x, y;
    char z [4];
} g;

int h;
.
.
.
```



Note:

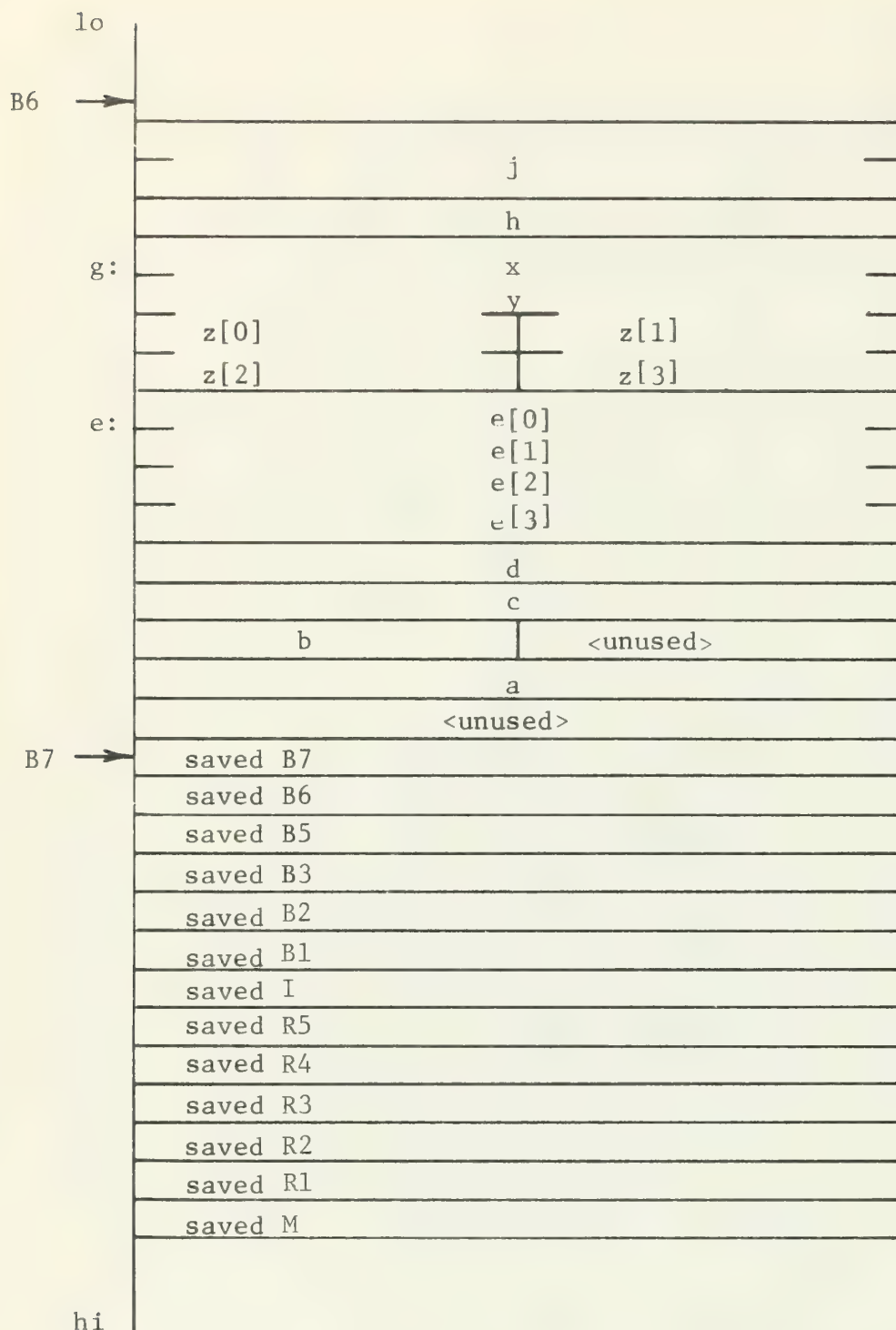
The value of c will be in R4.

The character variable b occupies only the low-order by of its word.

Storage for the array f will be in the data section of the system. The loader creates the data section as part of the loading process.

Snapshot of LSI-11 process stack with local variables

Figure 18



```

procedure ( )
{
    int a;
    char b;
    register int *c;
    int d;
    int e[4];
    static char f[1000];
    struct {
        int x,y;
        char z[4];
    } g;
    int h;
    char *j;
    .
    .
    .

```

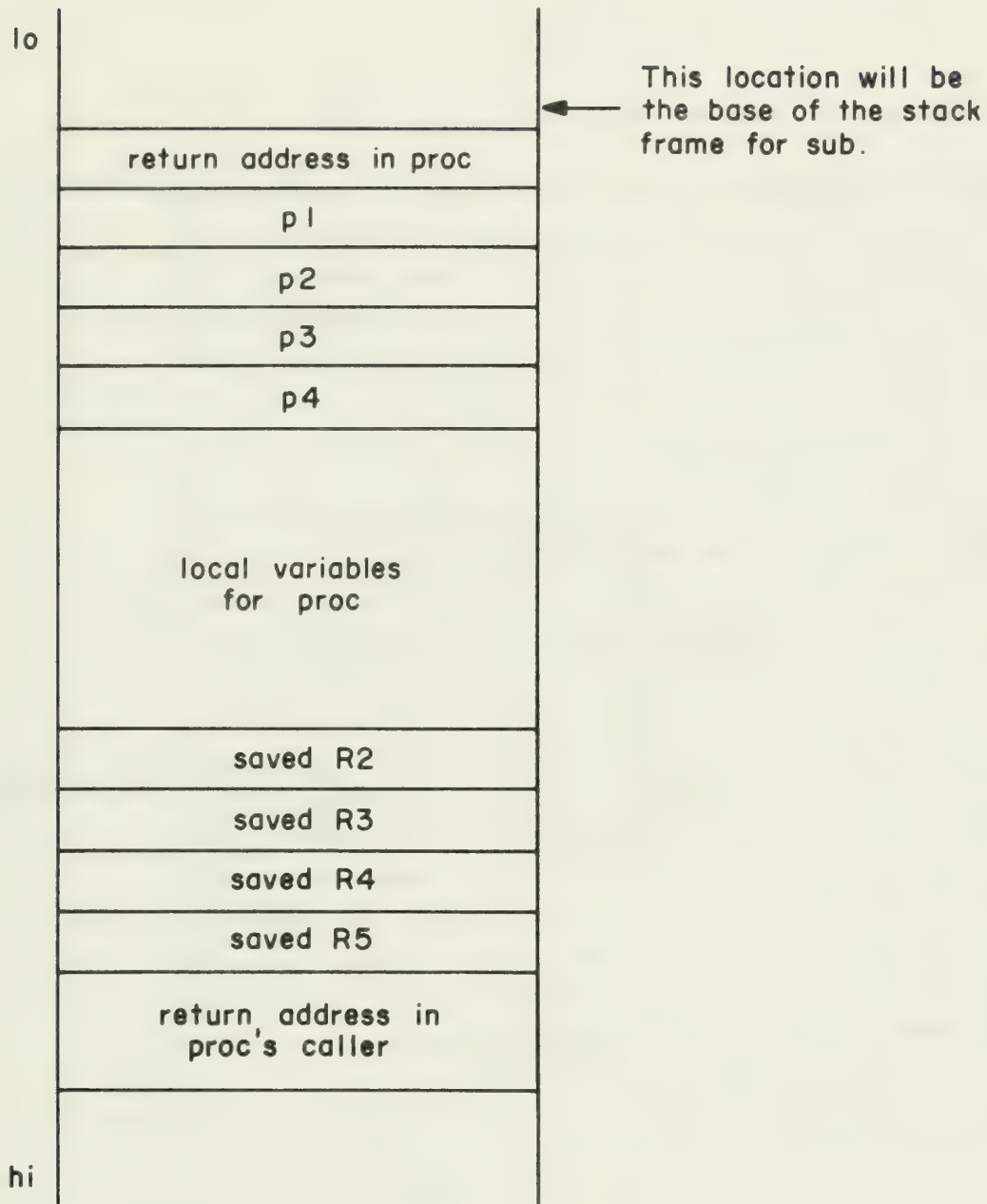
The character variable b occupies only the high-order byte of its word.

The register variable c is stored on the stack since Level 6 C does not utilize registers for user variables.

Storage for the array f will be allocated within the body of the system during the linking process.

Snapshot of Level 6 process stack with local variables

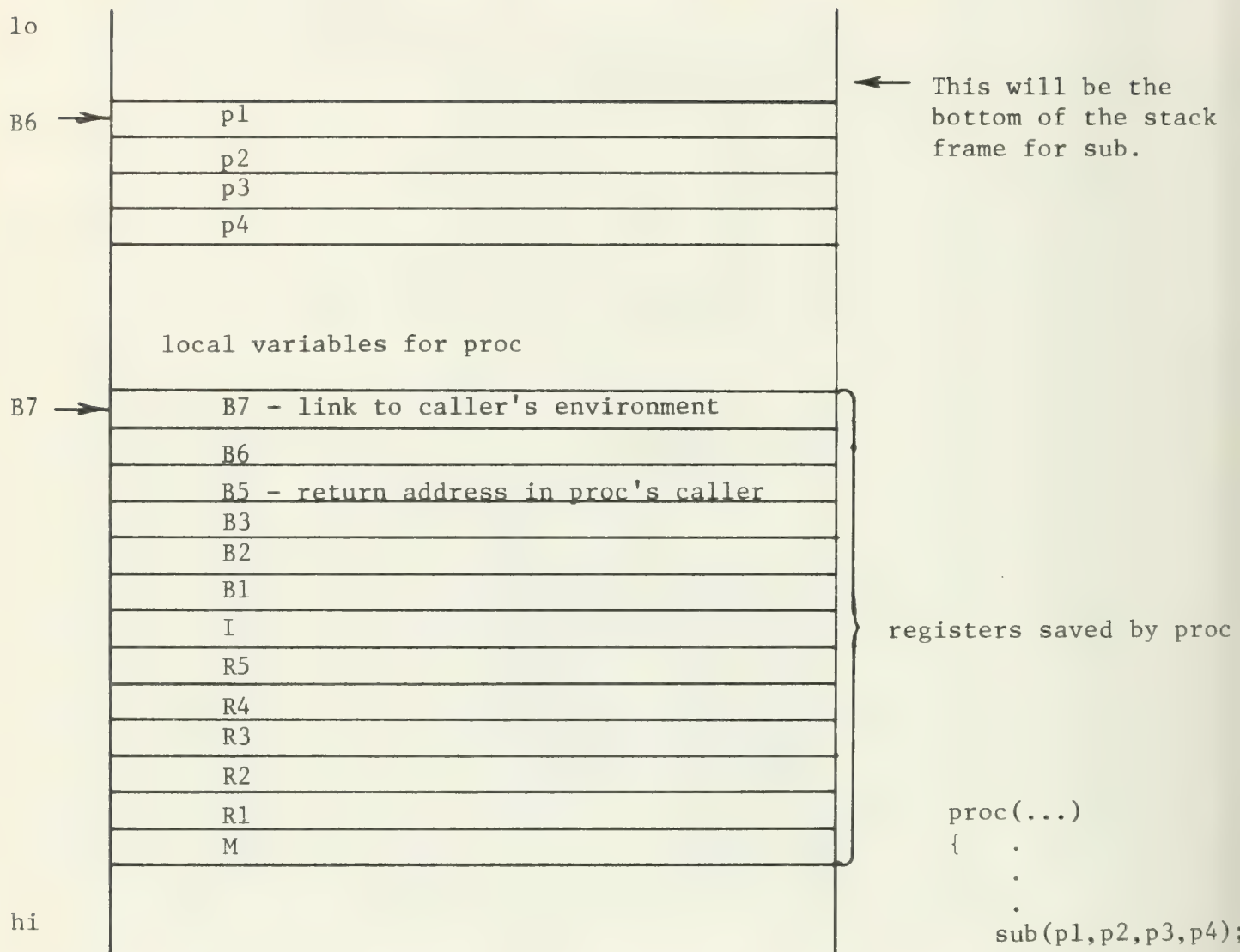
Figure 19



Snapshot of stack after call to subroutine sub by proc, and before sub has started execution.

Subroutine linkage - LSI-11

Figure 20



Snapshot of Level stack after call to subroutine sub by proc, before sub has started execution.

Subroutine linkage - Level 6

Figure 21

On the LSI-11, the calling procedure invokes the subroutine by doing a "jsr PC" to it. Upon completion the subroutine branches to cret which restores the stored registers and returns to the caller by doing an "rts".

On the Level 6, the calling procedure invokes the subroutine by doing a "LNJ \$B5" to it. Upon completion, the subroutine executes a "RSTR" instruction to restore its saved registers, and then returns to the caller by doing a "JMP \$B5".

Process Stacks. Once C's use of stacks is understood, the workings of process stacks on the IT is fairly straightforward. Stack manipulation occurs just as in standard C usage with each process using its own stack.

Each process stack on the IT has a five word header at the low address end of it. The data in this header is used by the scheduler when blocking and restarting the procedure, and is slightly different for the two types of terminals. On the LSI-11, the fields of the stack base, in order of increasing address, are:

1. the size of the stack in bytes,
2. the last R5 value for this process,
3. the last R6 (stack pointer) value for this process,
4. the priority of the process, and
5. a guard word.

On the Level 6, the fields, again in increasing order are:

1. the size of the stack in words,
2. the address at which the process should be restarted,
3. the last B7 (environment linkage pointer) value for this process.
4. the priority of the process, and
5. a guard word.

The guard word is set at process creation to 104401(octal)(hex 8901). If this value is ever changed, the system will assume that the stack has been overwritten somehow and will print an error message and halt.

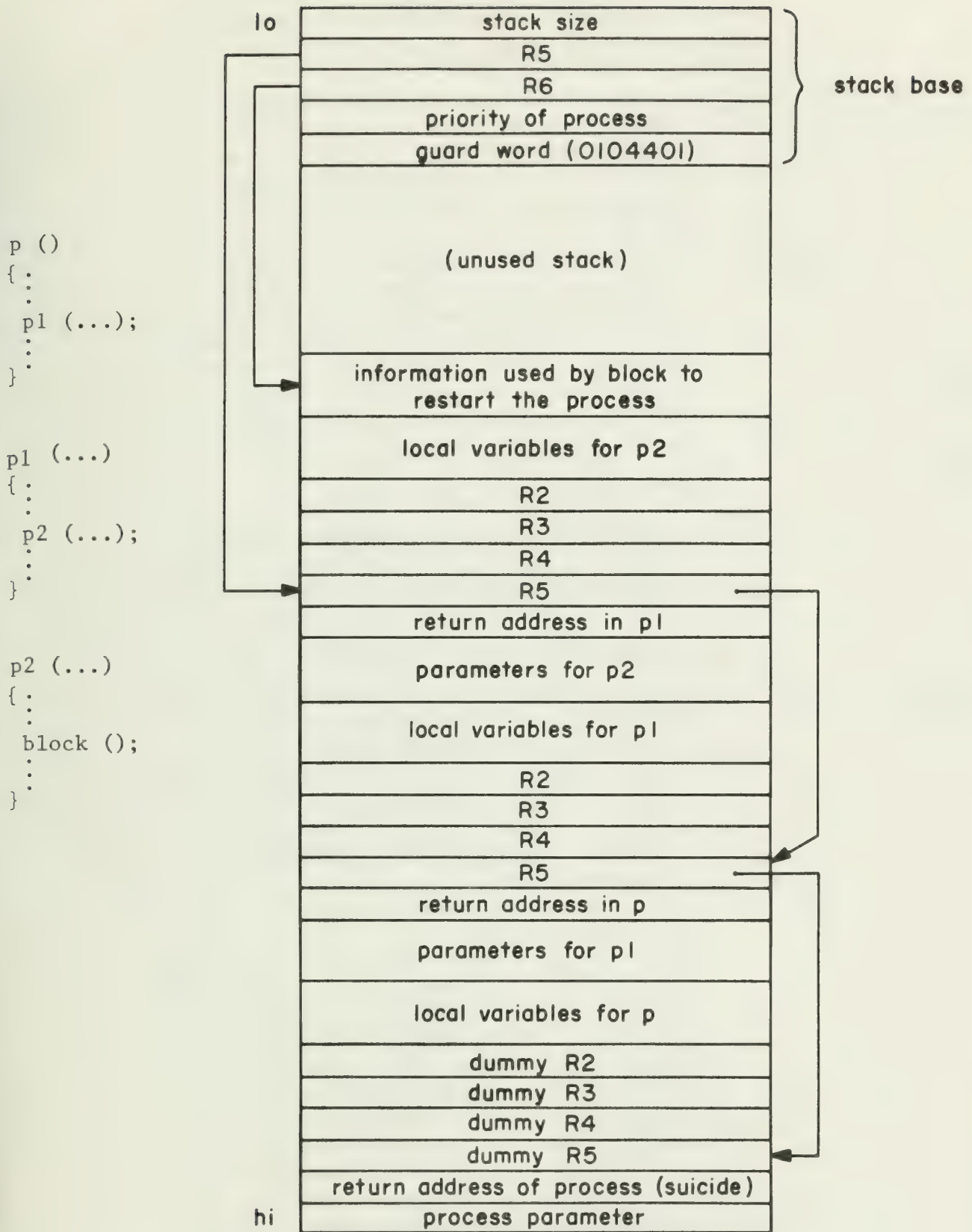
The bottom of the stack is set by creep to look as if the main routine of the process had been called by suicide. If the main routine ever returns, it will return to suicide which will perform the usual process termination actions.

Figures 22 and 23 diagram a complete process stack for LSI-11 and Level 6 systems respectively.

IT Post-Mortem

This section is an example of an investigation into an IT system. It is hoped that this will illustrate some of the points made in the preceeding discussion. Before starting, five points should be made.

1. This investigation deals specifically with an LSI-11 system. As such, the details such as the meanings of particular registers are not directly applicable to Level 6 systems. However, the general approach to investigating systems which is illustrated is applicable.
2. The system being investigated was intentionally stopped by the operator rather than crashing as a result of a system bug. Thus this investigation will be exploritory, to see what can be found out about the system, rather than bug-hunting.
3. The "discussion" will be more of a monologue, written in first-person. In any IT post-mortem, the specific approach used will vary widely between operators. The approach mentioned here is closely related to the approach evolved in the author's experience. Others may well find some other approach more useful for them.



Process stack - LSI-11 IT

Figure 22

```

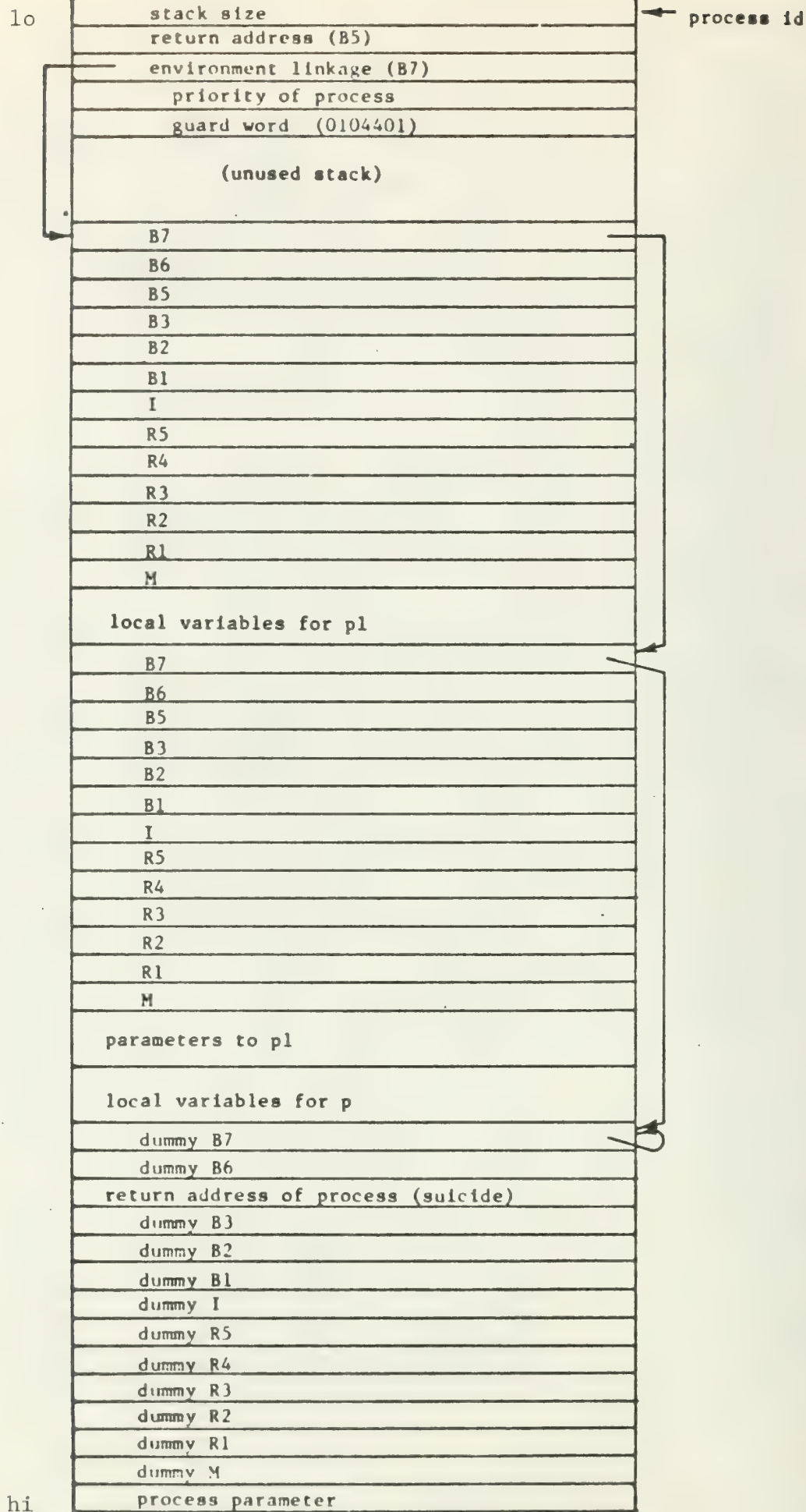
p ()
{
  .
  .
  .
  p1 (...);
  .
  .
}

```

```

p1 (...)
{
  .
  .
  .
  block ();
  .
  .
}

```



Process stack - Level 6 IT

Figure 23

4. All numbers mentioned are in octal. When listing memory locations and their contents, the form used will be "m:c", where m is the memory address and c is the contents of that location. Explanations as to the significance of a particular location or meaning of a value are included in parentheses.
5. Since the LSI-11 is a byte-addressable machine, the word-entries examined here always have even addresses. A similar dump of a Level 6 system which is a word-addressable machine, would contain odd-numbered word addresses. Also address of successive locations on the Level 6 increase by 1 instead of by 2 as on the LSI-11 system.

Before much meaningful information can be obtained from the post-mortem, we need a namelist to tell where things are. The namelist for this system is included on the next page.

Our first action is to check the register values. These are:

R0:	66772
R1:	00000
R2:	55334
R3:	64512
R4:	63234
R5:	67226
R6:	67216
R7:	30112
PS:	00000

R7, the program counter, will point to the first instruction after the instruction where the processor stopped. Comparing the contents of R7 to the namelist we see that the system died in read q. None of this is immediately useful, so we file it away and go on.

000400T	_diag	027306T	_set_mod	041622T	_mtps
000602T	_mk_curs	027476T	_write	041622T	_set_ps
000764T	_pr_n_cl	027666T	_io_init	041630T	_lrem
001400T	_sub	027772T	_pause	041646T	_ldiv
001464T	_t_async	030030T	_write_q	041662T	_exit
002072T	_t_graph	030072T	_read_q	041662T	_halt
003250T	_t_print	030126T	_vee	041666T	Bus_erro
003574T	_t_prior	030246T	_pee	041710T	Trap
004232T	_t_strin	030346T	_block	041742T	_fixup
005712T	_main	030426T	_first_b	042070T	csv
010004T	_t_touch	030522T	_enq_RQ	042104T	cret
010546T	_tt_ck	030740T	_enq	051136D	_tg_16
010724T	_t_ttp1	031076T	_deq	051176D	_tg_32
011542T	_t_ttp2	031236T	_alloc	051276D	_labels
012444T	_p3	031350T	_free	055264D	_PROCTAB
013670T	_t_ttp3	031604T	_error	055472D	_mess
014460T	int_disp	031674T	_tiod	056246D	Ascii_Ta
014574T	_startup	032064T	_tok_pri	056446D	_CS_8x16
015134T	_kb_driv	032612T	_ln_xpan	062524B	_PAGE
016222T	_kb_inte	033012T	_parm_xp	062524D	_edata
016276T	_ph_driv	033340T	_str_num	062534B	_tt_curr
017736T	_peeper	033532T	_get_cha	062734B	_ME
020034T	_phrint	033546T	_get_cur	062736B	_READY_Q
020106T	_phxint	033616T	_get_env	062744B	_FREE_PT
020164T	_tp_driv	033700T	_get_pag	062746B	_CORETAB
021104T	_tp_inte	033750T	_get_pg	063136B	_KBP_Q
021144T	_kb_echo	034024T	_get_siz	063144B	_PP_Q
021276T	_tt_sele	034054T	_mk_page	063152B	_TP_Q
021750T	_tt_read	034130T	_set_cha	063160B	_MAIN_Q
022276T	_tt_acti	034146T	_set_cur	063166B	_PH_addr
022410T	_tt_arra	034212T	_set_env	063170B	_PH_coun
023176T	_tt_clea	034274T	_set_pag	063172B	_DEV_TAB
023226T	_tt_crea	034340T	_get_tok	063332B	_CS_ID
023360T	_tt_dele	034472T	_verify	063334B	_curs_x
023464T	_tt_flas	034552T	_index	063336B	_curs_y
023626T	_tt_labe	034724T	_printf	064512B	_end
024374T	_tt_mark	035264T	_printn		
024554T	_tt_move	035374T	_put_asc		
024620T	_tt_outl	036302T	_putchar		
025146T	_tt_rela	036376T	_ring_be		
025224T	_scrunch	036444T	_area_li		
025276T	_creep	037256T	_put		
025432T	_suicide	037702T	_erase		
025474T	_kill	040326T	_putline		
025552T	_clear_i	041106T	_putdot		
025740T	_open	041166T	_screen_		
026306T	_close	041226T	_cmp		
026520T	_flush	041372T	_cvb		
026706T	_peek	041610T	_get_ps		
027104T	_read	041610T	_mfps		

System Namelist

Table 1

Remembering that PROCTAB has an entry for each process started at system initialization, we check the contents of PROCTAB:

(PROCTAB)	55264:	15134	(first word of first entry)
	55266:	00144	
	55270:	00000	
	55272:	00010	
	55274:	66772	
	55276:	16276	(first word of second entry)
	55300:	00144	
	55302:	00000	
	55304:	00010	
	55306:	67302	
	55310:	20164	(first word of third entry)
	55312:	00144	
	55314:	00000	
	55316:	00010	
	55320:	67612	
	55322:	05712	(first word of fourth entry)
	55324:	00764	
	55326:	00000	
	55330:	00006	
	55332:	70122	
	55334:	00000	(first word of fifth entry)
	55336:	00000	
	55340:	00000	
	55342:	00000	
	55344:	00000	(an entirely 0 entry, indicates last entry)

The entries in PROCTAB are process structures. These are explained in Appendix C. The first word of each five-word entry in PROCTAB is the starting address of the process' main procedure. The fifth word is filled in by startup to hold the process identifier, that is the address of the base of the stack. Decoding the appropriate entries we see:

<u>process</u>	<u>id</u>
kb_driver	66772
ph_driver	67302
tp_driver	67612
main	70122

Since this is an LSI-11 system, there is at least one process not included in PROCTAB. This is peeper, which is created by ph driver.

Generally, peeper's stack will be adjacent to main's. To caculate the address of the bottom (high address end) of main's stack, we get the size of main's stack in bytes and add it to the address of the stack.

The size of a process' stack can be found (in words) in the second word of the process' entry in PROCTAB, or (in bytes for LSI-11 systems) in the first word of the process' stack base. In any case we see that main's stack is 1750 (octal) bytes long. So we calculate that peeper's stack will probably start at 72072 (70122+1750). We check the locations starting at this address hoping to find a stack base. This consists of five words following the stack base structure described in Appendix C.

We find:

72072:	00144
72074:	72214
72076:	72202
72100:	177777
72102:	104401

This is a reasonable stack base, with the proper stack size and priority for peeper, so we will assume that the peeper process has an id of 72072.

Now we know the internal identifiers of all the processes. This will allow us to interpret the values in ME and the READY Q. Checking ME we find that ME is 66772, indicating that the keyboard process was running when the system died. Ordinarily we would remember at this point that the system was in read q when it died and would immediately narrow our investigation to the call to read q in kb driver to pinpoint the trouble. However in this exploratory example we will continue the investigation, to see what else we can find. Remember that we noted initially that the system was not killed by a bug.

Next we check the READY Q. Following the chain we find:

(<u>READY Q</u>)	62736:	64532	(pts to last element)
(last element)	64532:	72072	(id of peeper)
	64534:	64532	(pts to 1st element again)

In this case the READY Q has only one ready process, the peeper. At this point we have examined all of the generally useful system-wide variables. We will now look at the stacks and input queues of the various processes. This example will not show the entire stack for the various processes, since these tend to be fairly large and relatively uninteresting. The example will include the stack base and the R5 chain showing the sequence of procedure calls in the process.

We now check the phone process. The phone process input queue contains:

(PP_Q)	63144:	00000	
	63146:	177777	(-1 indicates 1 process waiting on q)
	63150:	64526	(ptr to list of waiting processes)

64526:	67302	(id of waiting process - ph_driver)
64530:	64526	(ptr back to this elt, closing list)

This queue indicates that the phone process is waiting on input data.

Checking the base of the phone process stack we find:

(base of pp stack)	67302:	00310	(size of stack, in bytes)
	67304:	67526	(stored R5)
	67306:	67512	(stored R6)
	67310:	00010	(priority)
	67312:	104401	(guard word)

All of these values are reasonable. The stack size, priority, and guard word are as they should be. These values are filled in by startup and should never change, so their value can be verified comparing them to parameters passed to creep for this process. The stored R5 and R6 values should both be larger than the address of the guard word (67312) and less than or equal to the address of the bottom of the stack ($67302+310-2=67610$ in this case). Also the stored R5 should be larger than the stored R6. (Note that addresses are treated as 16-bit unsigned numbers.) Since all the criteria are met, we will assume that this stack is in good shape and will check the R5 chain. We find:

(stored R5 points to:)	67526:	67542	(pts to previous R5)
	67530:	30112	(return addr in read_q)
	67542:	67604	(R5 link)
	67544:	16446	(return addr in ph_driver)
	67604:	67304	(dummy R5)
	67606:	25432	(beginning of suicide)

The reference to suicide is the dummy return address put in the bottom of the stack by creep during process creation. By tracing the list of return addresses we see that ph_driver called read_q which then called

something else not mentioned on the stack. Since the R5 chain only provides a list of return address when the R5 was also stored there will be no address in the chain for the current procedure. However there are several ways of deducing what was called. In the current situation we know the phone process was not running when the system died (ME is the keyboard driver), so it must have been blocked. If we guess that the last procedure called was block, then remembering that block uses a non-standard subroutine linkage so that the incoming R5 is not saved, we could guess that pee called block. This is an entirely reasonable sequence to assume since we know that read q initially calls pee and that the process will block if there is no data stored in the queue to be read. We can check our assumptions by inspecting the top word on the process stack. We find:

(stored R6 points to:) 67512: 30332 (return addr in pee)

Thus the entire calling sequence for this blocked process is the main routine ph_driver called read q which called pee which called block.

Inspecting the touch panel process' input queue and stack reveals a situation identical to the phone process'.

(TP_Q)	63152:	00000	
	63154:	177777	(-1 indicates 1 process waiting on q)
	63156:	64536	(ptr to list of waiting processes)
	64536:	67612	(id of waiting process - tp_driver)
	64540:	64536	(ptr back to this elt, closing list)
(base of tp stack)	67612:	00310	(size of stack, in bytes)
	67614:	70032	(stored R5)
	67616:	70016	(stored R6)

	67620:	00010	(priority)
	67622:	104401	(guard word)
(stored R6 points to:)	70016:	30332	(return addr in pee)
(stored R5 points to:)	70032:	70046	(R5 link)
	70034:	30112	(return addr in read_q)
	70046:	70114	(R5 link)
	70050:	20226	(return addr in tp_driver)
	70114:	67614	(dummy R5)
	70116:	25432	(beginning of suicide)

This indicates that the main routine tp_driver called read q which called pee which called block.

Since the keyboard process was running when the system died, its input queue and stack will be somewhat different than those for the other device drivers.

(KBP_Q)	63136:	64542	(ptr to list of input data)
	63140:	00000	(count field)
	63142:	00000	(no waiting processes)
	64542:	06401	(last element in list)
	64544:	64542	(ptr to same elt, indicating 1-elt list)

The keyboard process input queue points to a list of waiting data and has no list of waiting processes. However the count field (second word) of the queue is 0 rather than the 1 (for one data element) as might be expected. We will note this and continue the investigation. Checking the keyboard stack base we find:

(base of kbp stack)	66772:	00310	(stack size)
	66774:	67212	(stored R5)
	66776:	67176	(stored R6)

```
67000:    00010    (priority)
67002:    104401   (guard word)
```

which is not an obviously broken stack base. However we do not use the R5 and R6 stored in the stack base to investigate the keyboard stack. Since the keyboard process was running when the system died the stored R5 and R6 are out of date. They are only accurate when the process is blocked. For a currently active process an indeterminate number of procedure calls and returns may have been done since the last time the process was blocked. To get a handle on the stack in this case we use the values in the actual system registers R5 and R6. We noted before that these values are:

```
R5:  67226
R6:  67216
```

Chasing this R5 chain we find:

```
(R5 points to:)    67226:    67274    (R5 link)
                   67230:    15214    (return value in kb_driver)
                   67274:    66774    (dummy R5)
                   67276:    25432    (begining of suicide)
```

We know from this that the main routine kb driver had called some procedure but we don't know which one. Since this was the active process when the system died then the R7 register will hold the address of the instruction after where the system died. We noted before that R7 is 30112 which is in read q. So kb driver had called read q. Inspecting the instruction immediately prior to 30112, we find a call to pee. So we guess that read q called pee which waited until there was data enqueued on the keyboard process' input queue and then returned. Immediately after the return the system died. (This interpretation does in fact correspond to the way in which the system was actually killed by the operator.) At this point we return to that apparent inconsistency of a count of 0 for

a queue that has one data item on it. This results from a temporary condition occurring when the system died. The semaphore controlling the queue and its data has already been pee'd. Pee has decremented the count of elements in preparation for the element being removed, so the count is 0. However the system died before the data was actually removed catching the queue in an inconsistent state. (It should be noted that during execution all the queue-manipulating routines make themselves non-interruptable before operating on the queues to eliminate the possibility of an interrupt routine catching a queue in a similar state.)

The only remaining process of any interest in the main user process. This process does not use its input queue at all, so that MAIN_Q queue is entirely 0. We can perform the same sort of stack check on main as we did for the phone and keyboard drivers. We find:

(base of main stack:)	70122:	01750	(stack, size, bytes)
	70124:	71146	(stored R5)
	70126:	71132	(store R6)
	70130:	00006	(priority)
	70132:	104401	(guard word)
(stored R6 points to:)	71132:	30332	(return addr in pee)
(stored R5 points to:)	71146:	71162	(R5 link)
	71150:	27252	(return addr in read)
	71162:	71214	(R5 link)
	71164:	22036	(return addr in tt_read)
	71214:	71232	(R5 link)
	71216:	21322	(return addr in tt_selections)
	71232:	71610	(R5 link)
	71234:	13520	(return addr in p3)

71610:	71674	(R5 link)
71612:	14434	(return addr in t_ttp3)
71674:	72064	(R5 link)
71676:	07400	(return addr in main)
72064:	70124	(dummy R5)
72066:	25432	(beginning of suicide)

In this case we have a blocked process whose main routine main called t ttp3 which called p3 which called tt selections which called tt read which called read which called pee which called block.

PART III:

MAINTAINING IT SYSTEM SOFTWARE

OVERVIEW

Sections in this part of the manual provide detailed descriptions of certain sections of the IT system. They are designed for use by programmers who are responsible for the maintenance and improvement of IT system software. Application programmers should, in general, find the first two parts of the manual sufficient.

This part gives details on

1. the I/O system,
2. the file system, and
3. the Level 6 software interface to the remote display heads.

It is assumed that the reader is thoroughly familiar with the information contained in Part I.

MODIFYING THE I/O SYSTEM

The IT software currently supports a specific set of I/O functions and a specific set of external devices. For future versions of IT systems to support new operations or devices, IT programmers will need to modify the existing system. This section includes a discussion of how to add a new I/O function. Following that is a discussion of how to add a new device to the system.

These discussions assume that the new capabilities are to be added within the basic framework of the existing I/O system. This system is described in the System Overview and the I/O System and Device Handlers sections. The reader should be familiar with these sections before proceeding.

The source for the code referenced in this discussion can be found in the subdirectories devices, includes, io sys, kernel, and support.

Adding I/O Functions

Adding a new I/O function is a three step process:

1. Create the handling routine, e.g. read.
2. Modify existing device handlers to perform the new function.
3. Update the library /lib/libI.a to include the modified code.

Each of these steps is elaborated below.

Create the Handling Routine. To implement function X a handling procedure named X must be created. Typical parameters to X include:

1. the id of the device to perform the action,
2. a pointer to the caller's buffer,
3. the length of the caller's buffer, and
4. a pointer to a word in the caller's space to hold status information upon return.

Not all of these are needed for every function, of course, and some functions may require others. The read procedure is a good example of a simple I/O function handler.

The internal structure of X may be modeled on existing I/O functions such as read and write. This structure includes the following steps:

1. Verify that the caller has specified a valid device identifier. The device id should be a number between 0 and one less than the number of devices on the system inclusive. The number of devices is determined by the system define number_of_devices found in the include file constants.incl. This file should be included in the file containing procedure X so that number_of_devices is accessible.
2. Verify that the caller is the owner of the specified device. The id of the owner of each device is stored in the owner field of that device's entry in the system device table DEV_TAB. The identifier of the calling process is found in the system global ME. Thus the check that the caller owns the device compares ME to DEV_TAB[device_id].owner. If these are equal, then the caller owns the specified device.
3. Format a request block. This block is passed to the device handler which actually performs the necessary action. The request block is a standard system structure, and one block is associated with each device entry in the device table. This block is referenced as DEV_TAB[device_id].block. The type_req field of this block must be filled in, and other fields may need to be set depending on the function to be performed. It is good practice to zero any unused fields of the request block.

4. Send the request to the appropriate device handler. The request, represented by a pointer to the request block, is sent to the handler process by doing a write q to the handler's input queue. The address of this queue is also stored in the device table and can be found in `DEV_TAB[device_id].handler_q`.
5. P the semaphore associated with the request, blocking this process. This semaphore is found in

`DEV_TAB[device_id].block .req_semaphore`.

The device handler will read the request from its input queue. When the requested action has been completed, or if it is impossible, the handler will V the semaphore in the request block which will restart the requesting process.

6. Set return values and return to the caller. Typically the status word is set to 0 if the activity was successfully completed and non-zero otherwise. Meanings of non-zero codes may vary depending on the device. Additionally, a value such as the number of characters read or written is generally returned.

All of the I/O routines in the current IT system, except the special cases of open and create, follow this basic format. New routines of course may need to use a different approach. Careful consideration must be given to any new approach that would either:

1. omit P'ing the semaphore, or
2. allowing a non-owner to operate on a device.

Both of these constraints are closely tied to the system assumptions that all I/O is synchronous and that only one process at a time may access a device. Violating these assumptions may have catastrophic effects.

Modify Device Handlers. Current IT device handlers have a specific set of commands which they recognize as valid. If they receive an unknown command they mark the command's status word to indicate an invalid request and V the requesting process. In order to add a new command to the I/O system the handler for every device for which the command is appropriate must be modified to perform the new action. This is a two step process:

1. recognize the command, and
2. act on it.

To recognize a command, the device handler picks the `type_req` value from the request block and performs a switch statement on that value. If the value corresponds to one of the cases in the switch statement, then the command is recognized. To recognize a new command, a new case must be added to the switch. Currently, all cases are specified by labels such as "case `read_type`:" or "case `write_type`:" where `read_type` and `write_type` are system-wide defines included in `constants.incl`. A code for the new command should be added to `constants.incl`. The new I/O routine should use this code in formatting the request block and the device handlers should add a new case with this value.

The actions performed in the case will of course depend on the function to be implemented. At the very least the new case will need to be added, and the workings of the other cases may need to be changed to support the new command. Whatever changes are made, the device handler must always V the requesting process either when the action is completed or when it is determined that the action cannot be done. If the requestor is not V'd, it will wait forever for the command to finish.

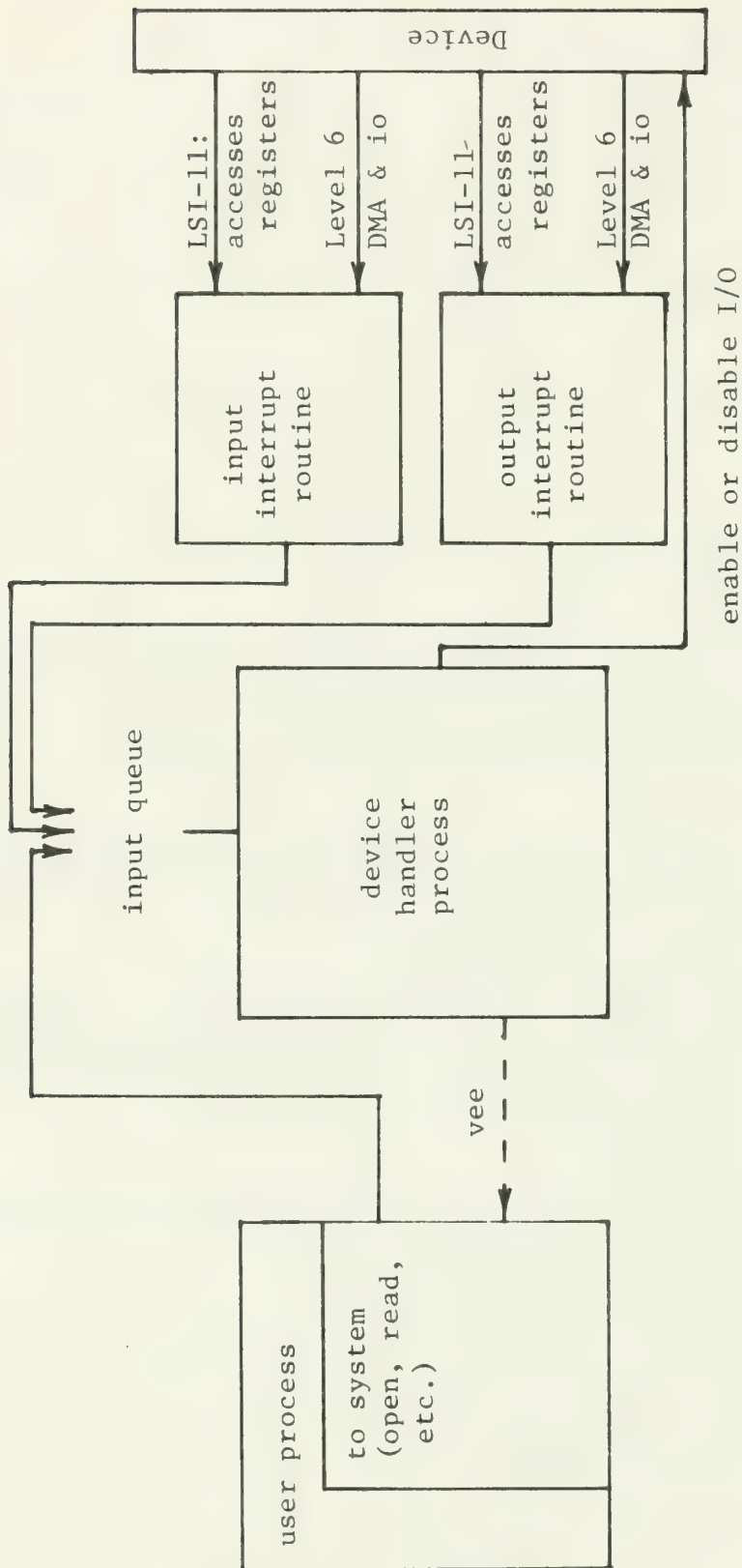
Update system library. After the new I/O routine and the device handlers have been debugged, they should be placed in the system library. For LSI-11 systems, this library is located in /lib/libI.a and is updated via the UNIX utility ar. The ar command is described in the UNIX Programmer's Manual [11], and the section Building An IT System includes a more complete discussion of building libraries. For Level 6 systems, the library is a disk file named lib whose members are object code for the system procedures. The object code for the new procedures should be copied into lib, replacing the old files for those procedures. See [12] for an explanation of the copy (cp) command in util13.

Adding a Device

To support a device new to the IT, several software changes or additions must be made. These changes are divided into the following categories:

1. Interrupt handlers are needed for the lowest level of transferring data from or to the device.
2. The I/O system must be modified to know of the new device.
3. A handler process must be written to interface the device to the I/O system.
4. The system initialization procedures must be modified to include the new device handler.

Each of these categories is described further below. It is assumed that the reader is familiar with the structure of the I/O system, the relationship to the I/O system of individual device drivers, and the interrupt mechanism of the LSI-11 or Level 6. The first two of these points are discussed in this manual in the section I/O System and Device Handlers. The LSI-11 interrupt structure is discussed in the DEC LSI-11 Processor Handbook [7]. The interrupt mechanism for the Level 6 is discussed in [6]. Figure 24 illustrates the flow of information between a device and the various portions of an IT system.



Flow of information and control between processes, interrupt handler, and device.

Figure 24

Interrupt Handlers. At the lowest level of the device interface are the interrupt handlers. These are C language routines which receive input data from the device and control sending output data to it.

The routines are interrupt driven, which means that they are invoked as a result of hardware generated interrupts rather than being called as normal subroutines. Currently, all devices on the IT except the disk are interrupt driven.

The structure of the interrupt routine will vary depending on the device, whether it is an input or output routine, and on the system for which it designed. The general structure of interrupt routines on the LSI-11 and the Level 6 is discussed below.

LSI-11 interrupt handlers: The general structure of an input interrupt routine is:

1. read the data from the device input register,
2. encode the data with a special value indicating input data, and
3. send the encoded data to the handler process for this device.

The method of encoding data will vary between devices. The methods used in the LSI-11 IT system all store the data in the high order part of a word and an odd input-data code in the low order part. All commands from interrupt routines to device handlers are odd. This is so they can be easily distinguished from pointers to request blocks, which are always word pointers and thus even.

The general structure of output interrupt routines is somewhat different. The output routine uses a global variable as a pointer to the buffer of characters to transfer and another global variable as the count of characters to send. Each time the routine is invoked it transfers one character and decrements the character count. If there are no more characters to transfer, the interrupt routines sends a message (a predetermined

odd number) to the handler indicating that the transfer has completed. For output devices such as printers, the interrupt routine may be modified to do any necessary character padding. This complicates the interrupt routine but significantly simplifies the device handler.

Typical examples of interrupt handlers in the current IT system are the phone line routines phrint and phxint.

Level 6 interrupt handlers: the general structure of an input interrupt routine on the Level 6 is:

1. copy the data from the interrupt handler's buffer to the driver process's buffer;
2. if the driver's buffer was previously empty, send a message to the driver telling it that there is now data in the buffer;
3. if not all of the data will fit in the driver's buffer, send a buffer overflow message to the driver and ignore the rest of the data in this buffer.

The driver process uses a cyclic buffer, indexed by global variables X_next and X_free to indicate the next data character and next free slot respectively. X varies depending on the specific driver, for example PH_next or KB_next. Only the interrupt routine adds data to the buffer, and only the driver removes data. By carefully determining the order in which data is inserted and indices are incremented, two different routines can access the buffer with no synchronization problems.

The general structure of output interrupt routines on the Level 6 is different both from input routines and from LSI-11 output interrupt handlers. On the Level 6, the device process accesses the device directly to initiate the transfer. Since all devices have DMA capabilities, they can then operate unattended until the transfer is complete. At that point the output interrupt routine is invoked. The

interrupt routine merely sends a message to the device process which then turns off output from the device.

Typical examples of interrupt handlers for the Level 6 are phrint and phxint.

Additional modifications - LSI-11: On LSI-11 systems, addition to the device interrupt routines, the system general interrupt handler and the device interrupt vectors must be updated. This requires some understanding of how interrupts are handled on the IT. As on all LSI-11s, each device has two words of low memory allocated as its input interrupt vector and two additional words for output interrupts. The values for these memory locations are determined by the contents of the object file low.o. When the device causes an interrupt, the LSI-11 performs an interrupt sequence automatically. This includes saving the current value of the program counter (PC) and the processor status word (PSW), and setting them from the two values stored in the designated interrupt vector. In the IT all interrupt vectors use the same value for the new PC. This is the address of the general interrupt handling procedure, int disp. Int disp performs actions such as saving current register values, then invokes the appropriate handler for the interrupt. The PSW value set from the interrupt vector is used to specify which routine to use. In particular, int disp keeps a table of interrupt handlers in a specified order. The interrupt vectors are set up with a PSW value of 340 (octal) plus the code for the index into the table of device handlers. For example, the input interrupt handler for the phone, phrint, is the fourth entry in the array of handlers, so the PSW part of the interrupt vector is set to 344 (octal). The order of handlers in the int disp table must correspond to the values in the interrupt vectors set in low.o.

To update the general interrupt routine to handle the device:

1. Modify low.s by setting the interrupt vector values in the vectors used by the device. The low order four bits of the PSW value will be used as an index into the int disp array of interrupt routines.
 2. Modify the number of routines defined in int disp (the equate "table_size") to include the new device.
 3. Add the names of the interrupt routines for this device to the table handler in int disp.
 4. Add the priority at which the interrupt routines should run to the table priority in int disp. The priority entries must be in the same order as the handler entries. They are used to set the priority of the interrupt routine as it is invoked.
- These changes will allow the new device to be used in an interrupt driven mode.

Additional modifications - Level 6: On Level 6 systems, in addition to creating the device interrupt routines, an interrupt vector must be added to low for the new device. This can be done by following the example of one of the entries in low. However it is somewhat easier if the interrupt mechanism on the Level 6 is understood. Every interrupt on the Level 6 will occur at a specific hardware level, call it n. If n is less than the level at which the machine is currently operating the interrupt will be honored, otherwise it will be queued for later attention. When the interrupt is honored, the hardware will pick up the pointer to the interrupt save area for level n. This value is stored in memory location 80 (hex) plus n, and points to an area containing between five and twenty-one words. The meaning of the first five words of the interrupt save area is:

1. a word which is used by the hardware to store the identifier of the interrupting device,
2. a bit mask which determines which, if any, of the registers are to be stored,
3. one word reserved for future use,
4. the address at which execution for this level is to begin, (in IT systems, this word initially contains the starting address of levlp), and
5. the new status word for this level (the only value which is changed in the status word is the privileged mode bit).

The next sixteen words (maximum) are used for storing registers, according to the mask in word 2. As processing begins at a new level, the general registers are loaded with the values in the register save area as part of the level changing mechanism. The initial values of the registers are stored in the interrupt save area of the preempted level. The program counter and status word are loaded from words 4 and 5 of the save area, and execution proceeds at the new level. When execution at the level terminates, either because of a higher priority interrupt or because the level voluntarily relinquishes the processor, then the general registers, the program counter and the status word are stored in the save area. A fuller explanation of interrupt on the Level 6 as well as the use of levels is contained in [6].

In addition to the five values in the interrupt save area, the register storage area must be initialized properly in order to work with levlp. Upon entrance, levlp assumes that

1. B3 contains the address of the interrupt handler for this level, and
2. B6 points to the stack for the interrupt process to use.

All other registers should have values of zero to avoid problems. Also, the word before the first word of the save area should be set to zero. This word is used for handling traps at this level.

To modify low to support a new device, the programmer must take the following steps:

1. reserve space for the interrupt stack,
2. create an interrupt save area with the proper values filled in, and
3. point the interrupt vector for the level at which the new device will interrupt at the new interrupt save area.

The values in the interrupt save area should all be zero except:

1. the save mask should be all ones (FFFF hex) so that all the registers are saved,
2. the new PC should point to the beginning of levlp,
3. the new status word should be 4000 (hex),
4. the saved value of B6 should point to the bottom (large address end) of the interrupt stack, and
5. the saved value of B3 should point to the procedure to use for handling the interrupt.

The structure of low.a can serve as a guide when adding new devices. In following the actions caused by handling an interrupt, it is important to remember two details:

1. the interrupt procedure is a standard C routine which saves and restores register value upon invocation and return. Thus the value of B3 upon return still points to the beginning of the interrupt handling procedure.

2. when processing terminates at the interrupt level, the registers and PC will be saved in the interrupt save area. This leaves the stored PC pointing at the instruction in levlp after the LEV. Accordingly, this instruction is a branch to the beginning of levlp, so that later interrupts can also be processed.

The procedure levlp does not require any modification to support new devices.

I/O System Modifications. The I/O system must be modified to know about the new device being added. Primarily this involves creating an entry in the device table DEV TAB for the device and initializing this entry. To do this:

1. Change the define number of devices in constants.incl to be the number of devices actually supported on the IT including the new device.
2. Update the procedure io init to initialize the DEV TAB entry for the new device.

These changes will allow the I/O system utilize the new device, under the assumption that the device has a standard device handler.

Device Handler. The new device must have a standard device handler process associated with it. This process will receive commands from the I/O system and the interrupt handlers for the device. It is the function of the device handler to correlate the actions of the device via the interrupt routines with the requests of the I/O system.

Device handlers on the IT are very similar in structure. Each of them do some initialization such as enabling input interrupts and setting up buffer pointers, then enter a never-ending loop. Inside the loop, the following actions are taken:

1. The handler reads from its input queue. If there are no commands in the queue, the handler will block at this point, and will remain blocked until something is added to its queue.
2. The data read from the queue is decoded. On the LSI-11 IT, if the data is an even value, it is taken to be a pointer to a request block. The command to be performed is found in the `type_req` field of the block. If the value is odd, it is taken to be a specially encoded one word value from the interrupt routines. (Odd values are useful in this situation since request block addresses will always be even.) On the Level 6 IT, the data is compared against a set of low-numbered pre-defined codes. If it matches one of them, then it is taken to be the command. Otherwise, the data is taken to be a pointer to a request block containing the command. The encoded command is translated into the corresponding command value in the handler, and any input data is also decoded.
3. A switch is performed on the decoded command.

The switch will cause specific actions to be taken depending on the command to be executed; however, some interrelation between commands is often necessary. For example, if there is not enough data in the local buffer to fulfill a read request, that request will be saved. As data is input later, the "data" case will need to V the reading process when the read is completed. The phone handler ph_driver may be useful in illustrating ways of handling the interdependencies of some cases.

Most cases are straightforward. However, the act of starting an output transfer on the LSI-11 deserves a word of explanation. When the handler receives a request to write data, it sets two global variables. One of these points to the user's buffer of characters to be output and

the other is the number of characters in the buffer. The handler then enables output interrupts for the device and notes that it has started a write. The enabling of output interrupts will automatically cause the hardware to generate an interrupt which will invoke the output interrupt routine as described above. This routine then transfers the characters in the buffer to the device. This transfer is transparent to the handler process and is asynchronous to it. When the last character is written, the routine will be invoked as usual. Noting that the global character count is 0, the interrupt routine will send a done command to the device handler. Upon receipt of the done, the handler will disable output interrupts and V the writing process.

System Modification. After all the above changes have been made, a few system changes need to be made to fit all the pieces together. These include:

1. The contents of the file PROCTAB.c must be updated to include the definition of the handler queue for the new device handler process and the contents of the structure PROCTAB must be updated to include the new handler as a process that will be created at system initialization.
2. The name and id of the new device must be added to the constants.incl file. This will be a define of the form

```
#define DEVICE "/dev_device"
```

```
#define DEVICE_LOC n
```

where n is an integer. The defined ids of the devices must be assigned so that each device id is greater than or equal to 0, is less than the number_of_devices value, and so that each device number is unique. The names of all devices on the IT are character strings that begin "/dev_".

3. The system array dev names must be updated to include a pointer to the name of the device. This entry should have the form &DEVICE and be added anywhere before the entry for DISK0.
4. The array dev types must be updated. The order of elements in dev types must correspond to the order in dev names. An entry whose value is -DEVICE_LOC -1 must be added in the position corresponding to the new entry in dev types.

When the above changes have been made, all the new routines, all modified routines, and all the I/O functions must be recompiled. When these new routines have been tested and debugged, they should replace the old versions of routines in (or be added to) the system library.

The portion of the I/O system which supports disk files is very different from the rest of the I/O system. This section describes the code which directly relates to disk files on the IT. The source for the code described here is contained in the directory disk.

Before the discussion of the disk software, a subsection is included to describe the physical and logical structure of floppy disk files.

Disk Format

To understand the workings of the IT file system, it is necessary to have an understanding of the organization of the files on disk. This includes:

1. the physical structure of a floppy disk,
2. the relationship between that structure and the logical structure used by the file system,
3. reserved sections of the disk,
4. the logical structure of files, and
5. the logical structure of a directory.

Each of these points is discussed below.

Physical Structure of Disk. A floppy disk is broken into sectors of 128 eight bit bytes. The sectors are organized into tracks. Each disk has 77 tracks of 26 sectors each. Tracks are numbered 0 through 76, and sectors are numbered 1 through 26. This is compatible with the IBM 3740 floppy disk format.

Two sectors of the disk are reserved for bootloader programs. Track 0, sector 1 contains the bootload program for the AED disk drive

used by the LSI-11 IT. Track 1, sector 1 is reserved for the bootload program for DEC drives. (Previous versions of the LSI-11 IT have used DEC drives, so all disks have both bootloaders for compatability considerations)

There is an additional discrepancy in the way the disks are accessed by the AED and DEC drives. The AED drive reverses the order of the data bytes in a word as compared to the order used by DEC. To nullify this effect, the sector read and write routines (s read and s write) for ITs with an AED drive introduce an additional reversal of the data bytes read or written.

Logical Structure of Disk. To the file system, the block is the basic unit of the disk. A block is composed of four logically contiguous logical sectors. A logical sector is equivalent to a physical disk sector except in their ordering. Logical sectors are mapped onto physical sectors in a manner designed to minimize the time lost due to rotational latency in accessing sequential logical sectors.

Disk blocks are numbered starting at 0. Block 0 corresponds to the first four logical sectors of the first track of the disk.

Reserved Blocks. Three disk blocks are reserved for special functions on the disk. These are:

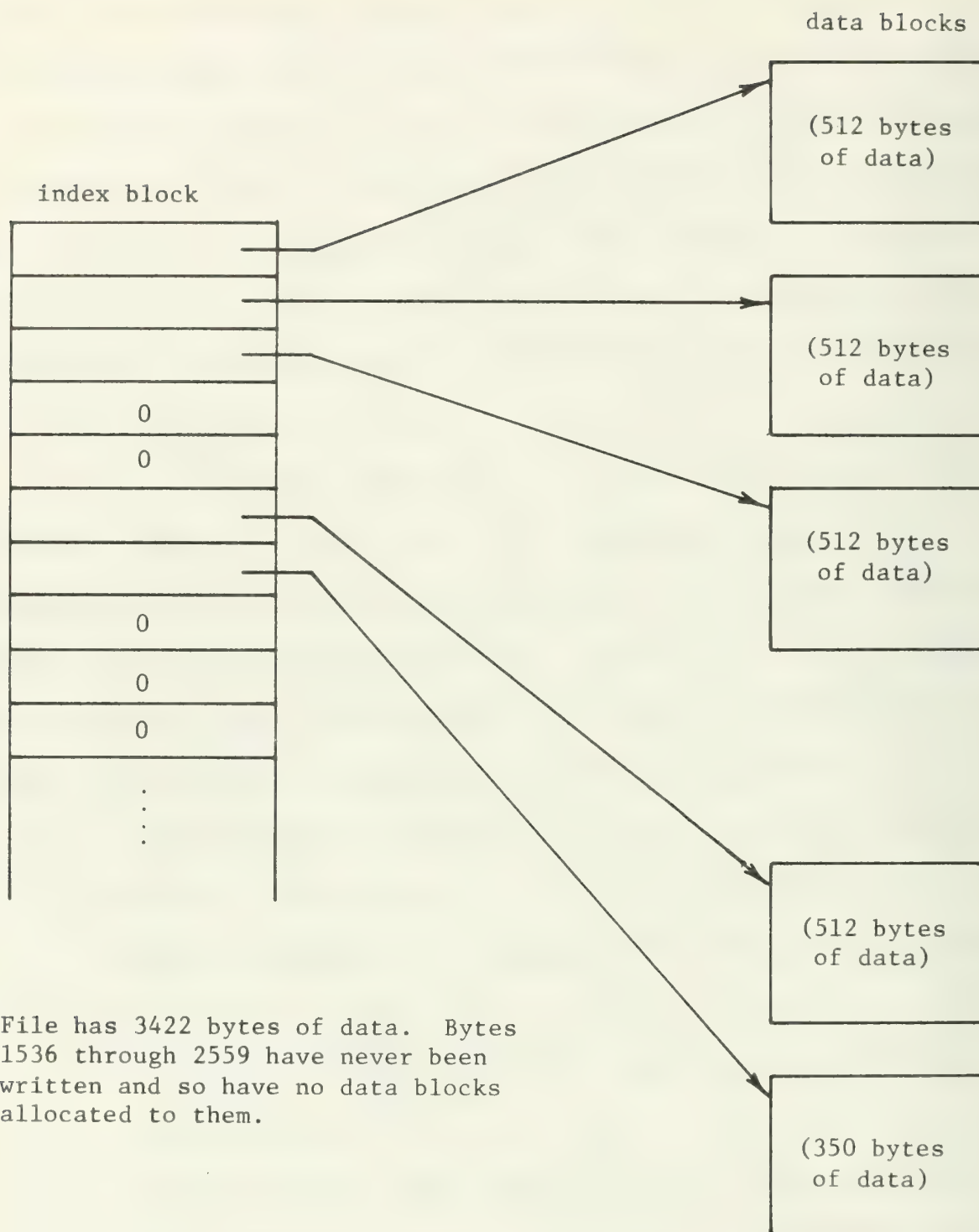
1. Block 0 contains the bootloader for LSI-11 ITs with an AED disk drive.
2. Block 4 contains the index block for the main directory for the disk. The meaning of index blocks and the structure of a directory is explained below.
3. Block 6 contains the table of unallocated disk blocks (the freemap) and the bootloader for LSI-11 ITs with a DEC disk drive.

File Structure. IT files are stored as indexed files on the disk. The contents of each file are stored in a set of blocks. An additional block, the index block, is used to allow the file system to find the blocks containing the file in the proper order. The index block contains 256 one word (two byte) entries. The n^{th} entry in the index block is the number of the block containing the n^{th} piece of the file. Zero entries in the index block are used to indicate portions of the file that have never been written, either because the file is not large enough to need the blocks or because the file has an internal hole. Figure 25 illustrates the structure of a file.

Directory Structure. A directory in the IT system is merely a file with a known format. It is stored as a regular file and is composed of a series of 32 byte entries. These entries are `dir_entry` structures and describe the files contained in the directory. The structure of the directory entries is explained below followed by notes on special conventions used in the directory. Figure 26 illustrates a disk directory entry.

Directory entry: The `dir_entry` structure is detailed in Appendix C. In general, the elements of a directory entry are:

1. One word for the type of the file. This is 0 for regular files, and non-zero for directories.
2. Three words of filler. These are currently unused.
3. Two words encoding the size of the file. The first word is the number of data blocks in the file minus 1. The second word is the offset in the last block of the last byte in the file.
4. One word containing the number of the index block for the file.



Structure of file

Figure 25

dir_entry:

pointer to parent's index block for directory files; 0 for data files
(unused)
one less than number of data blocks in file
number of bytes of data in last data block
pointer to index block for file
name of file (null terminated) (17 bytes)
flags (1 byte)

Structure of a directory entry

Figure 26

5. Seventeen bytes containing the name of the file. The name can be at most sixteen characters, and will be followed by an ASCII NUL (octal 0).
6. A one byte flag, used to indicate empty entries in the directory or files that are not to be deleted.

The `dir_entry` values contain all the information needed to access a file.

Special conventions: Some special conventions are followed in the directory elements. These are:

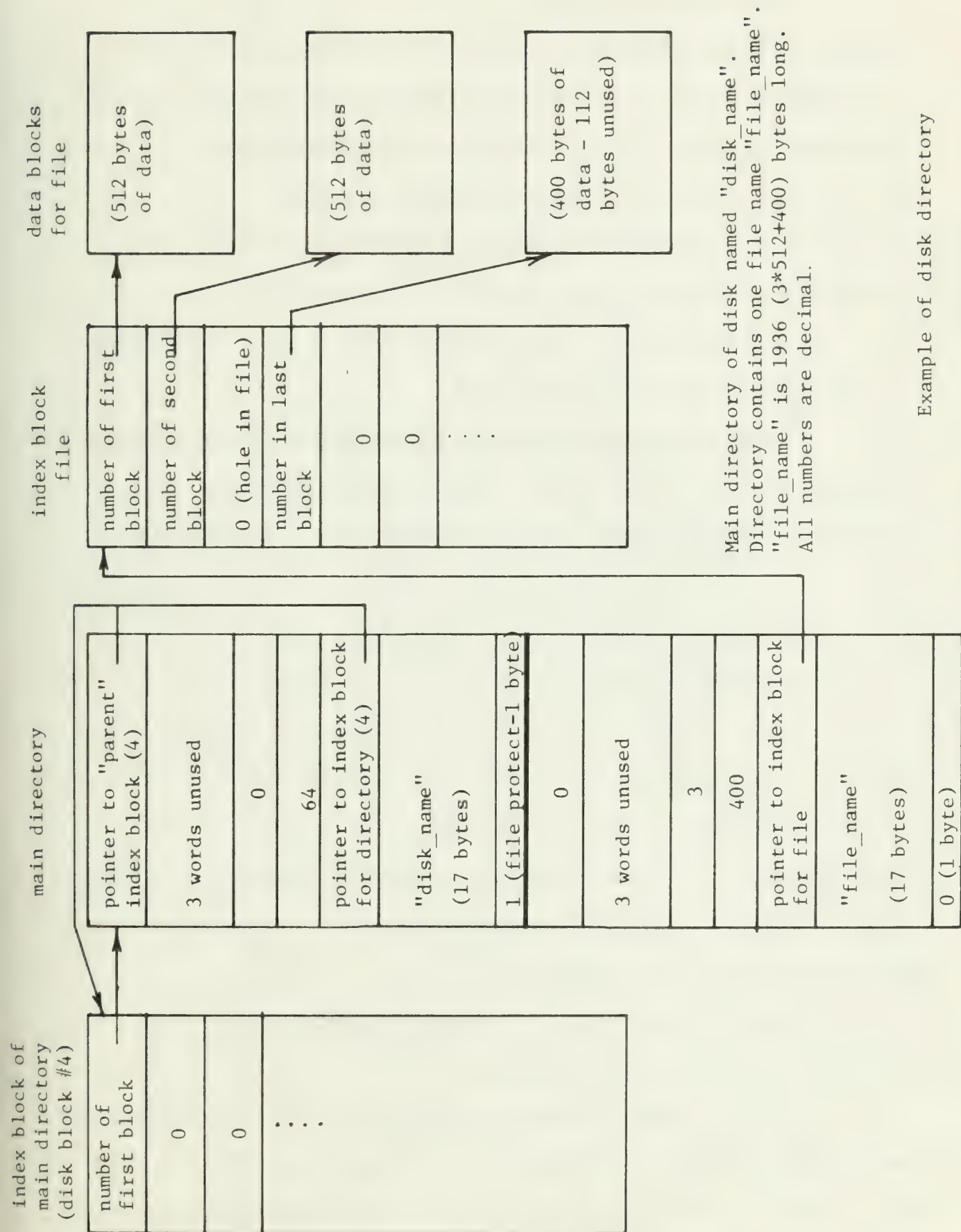
1. The first entry in the directory describes the directory file itself.
2. The type code for a directory's entry for itself is the number of the index block of its parent directory. This allows the directory structure to be traversed both upwards and downwards. It should be noted that the root directory for the disk points to itself as its parent.
3. The information in a directory's entry for a subdirectory is, in general, only accurate so far as the type, index block number, and name are concerned. All other information, such as the size of the subdirectory file, are contained in the subdirectory's entry for itself.

Figure 27 shows an example of a directory with one file.

Code Structure

The code relating to the IT file system is broken into six categories. These are:

1. the I/O system,
2. the disk driver process,
3. primary file level routines,



Main directory of disk named "disk_name".
 Directory contains one file name "file_name".
 "file_name" is 1936 (3*512+400) bytes long.
 All numbers are decimal.

Example of disk directory

Figure 27

4. file level support routines,
5. block level routines, and
6. sector level routines.

Each of these categories are explained below. The procedures and structures discussed are detailed in Appendix B and Appendix C of this manual respectively. The source code for these procedures is included in the UNIX subdirectories io sys, devices, and disk.

I/O System. The I/O system is described in detail in the section I/O System and Device Handlers.

Disk Driver. The disk driver process is described in the section I/O System and Device Handlers.

Primary File Level Routines. There are five main file level procedures in the IT file system. These roughly correspond to the primary user functions that can be performed on files. The routines are:

1. fopen,
2. fclose,
3. fio,
4. fseek, and
5. fdelete.

These procedures call the file level support routines and the block level routines to perform the functions indicated. Each of these five procedures is described below.

Fopen: Opens a file on disk and initializes a file_info structure to allow access to the file.

Fclose: Writes out the in-memory image of the data buffer for the file and the directory element for the file. After this is done, the in-memory information about the file can be thrown away, since all relevant data has been written to disk.

The buffer and directory entry for the file are only written out if they have changed since they were read into memory. The need to update the file contents on disk is indicated by the `bfflag` in the buffer structure for the file being set to 1 (the defined value of `dirty`). The directory entry needs to be updated if the `fflags` entry in the `file_info` structure is set. If this flag is set, the new file length is written directly to the directory entry.

`Fio`: This routine performs the standard input and output to a file. The operation of `fio` is governed by a flag that indicates whether it should

1. read data,
2. write data, or
3. flush internal buffers.

Before taking any action specific to the desired function, `fio` performs some general setup functions. In particular

1. the read/write position in the file is evaluated to determine the number of the file block containing the requested position and the offset into the block.
2. If the portion of the index block in memory does not have the entry for the required block, then the current subset of the index block is written to disk if necessary, and the appropriate subset is read into memory. The index block subset needs to be written to disk if the `ndx_f` entry in the `file_info` structure for the file is set.

After this initialization, the action of `fio` depends on the function to be performed. Each of these are described below.

On input, if the specified length of the read would cause the read/write pointer to move past the end of the file, the length of the read is truncated to be the actual amount of unread data in the file.

The request is then partitioned into one or more sections, each of which is contained in one disk block. The requests for reading the sections are passed to bufio which handles actually transferring the data to the user's buffer.

Attempts to read past the end of a file will return a data count of 0, after filling the user's buffer with 0's. If the user reads data from a hole, the appropriate data count is returned but all data from the hole will appear to be 0.

Writing is similar to reading in that the request is broken into sections on block boundaries and then bufio is used to actually do the transfer. In addition fio will handle allocating disk blocks to a file when new sections of the file are written. These new sections can be allocated either at the end of a file or in a hole.

Doing a flush through fio will clean up the file_info structure for the file by causing the in-memory version of the index block to be written out, if necessary.

Fseek: This routine changes the read/write pointer for the file. The entries fblkno and foff in the file's file_info structure are updated to specify the new position. Fseek does not cause any data to be read from or written to the disk. Any such needed updates will be done automatically the next time the contents of the file are accessed.

Fdelete: Deletes a file from disk. The file is flushed via fio, and then truncated via ftrunc. The in-memory copy of the disk's freemap is updated to include all the space previously occupied by the file. The directory containing the entry for the file to be deleted is then opened and the flgs element of the file's entry is set to the defined value empty_entry. If every step succeeds, the memory version of the disk freemap is then written to disk.

File Level Support Routines. The primary file level routines are supported by nine other routines that manipulate data at the file level. These are:

1. blk_alloc,
2. blk_free,
3. init_fib,
4. init_free,
5. dir_open,
6. ftrunc,
7. pathname,
8. save_free, and
9. xopen.

The function of each of these procedures is described in Appendix B.

Block Level Routines. There are three procedures that manipulate logical disk blocks. These routines

1. bufio,
2. zero_blk, and
3. zero_sim

know nothing about either the logical structure of the file system or the physical configuration of the disk. They are used by the file level procedures for doing block level I/O. They use the s read and s write routines to actually perform the I/O. Each of these procedures is discussed below.

Bufio: Performs block level input and output. This procedure is analogous to the file level routine fio. Where possible, bufio will transfer data directly between the user's buffer and the disk. If the user's request is less than a sector long, this is not possible so the bfstore array in the buffer structure associated with the file is used.

When the data in the `bfstore` array must be overwritten, the previous contents are written to disk if necessary. Such an update is necessary if the data in the `bfstore` has changed, which is indicated by the `bfflag` in the buffer structure having the defined value `dirty`.

Bufio will handle any amount of data from one byte through a full block. It has three modes of operation, explained below:

1. `read`: Transfers data from the disk or the `bfstore` array to the user's buffer.
2. `write`: Transfers data from the users buffer to the disk or the `bfstore` array.
3. `flush`: Cleans up the buffer structure by writing out the `bfstore` array if the `bfflag` is set to `dirty`.

`Zero blk`: Writes zeros to an entire disk block. This is used to clean up a block before allowing it to be allocated to a file.

`Zero sim`: Simulates bufio when the user is reading a hole. Fills the user's buffer with zeros.

Sector Level Routines. The lowest level disk routines deal directly with the disks. They cause one sector of data, the only amount the drives will accept, to be read or written. Two procedures provide these functions:

1. `s_read`, and
2. `s_write`.

These routines transfer data between the user's buffer and the disk. Additionally, they perform some mappings to improve the speed and compatability of the disks. In particular s_read and s_write:

1. map logical sector numbers into physical sector numbers. This is done to reduce the time lost to rotational latency when accessing logically sequential sectors.

2. reverse the order of bytes written as described in the section titled Disk Format.

It should be noted that by replacing the s_read and s_write routines by procedures designed to interface to a different drive, the entire file system will operate on an IT with that type of disk drive.

ACCESSING REMOTE DISPLAY HEAD

This section describes the Level 6 IT software used to access the plasma panel, touch panel, and keyboard of the remote display heads on these terminals.

One or more of these heads (up to a maximum of three) comprise the interface to the user of a Level 6 IT. These heads each consist of:

1. a parallel plasma panel,
2. a touch panel, and
3. a keyboard attachment.

One keyboard is provided with the system, and this keyboard can be attached to any of the remote display heads.

Each of these heads is controlled by a Zilog Z80 microprocessor. The interface to the Level 6 is through one general purpose interface (GPI) port on a Multiple Device Controller (MDC) interface board. Further details of this interface are included in [5]. Application programs can read data from the keyboard or any of the touch panels, and can display output on any combination of the panels.

Driving the Plasma Panel

Graphic operations and text displays are performed from the Level 6 processor by sending commands to the plasma panel controller. A command consists of a task code, which specifies the type of operation to be performed, and optional parameters. A list of valid task codes and their parameters is given in Table 2.

Application routines will normally access the plasma panel thru the panel graphics routines (putdot, putline, etc). These routines each format a local buffer with parameters specific to the desired operation. They then call the routine pp_write with the task code for the operation and a pointer to this buffer.

Task Word	Operation	Parameters
00	No operation	
01	Erase page	
02	Ring bell	
03	Enable keyboard input	
04	Enable touch input	
05	Disable keyboard input	
06	Disable touch input	
10	Read keyboard	
12	Read touch panel	
20	Erase dots	x,y,...
21	Write dots	x,y,...
22	Erase many vectors	x,y,vector,vector,...,vector
23	Write many vectors	x,y,vector,vector,...,vector
24	Erase single vector	x,y,vector,count,...
25	Write single vector	x,y,vector,count,...
26	Erase line	xbegin,ybegin,xend,yend,...
27	Write line	xbegin,ybegin,xend,yend,...
28	Erase box	left,bottom,width,height,...
29	Write box	left,bottom,width,height,...
2A	Erase outline	left,bottom,width,height,thickness,...
2B	Write outline	left,bottom,width,height,thickness,...
2F	Write characters, erasing ahead one line	character, character,...
30	Write characters	character,character,...
31	Read cursor	
32	Set page	z80 page descriptor
33	Set cursor	x,y (in dots)
34	Set charset	width,height,effector variable,charset variable
43	Init z80 variables	none
44	Request space for a variable	variable, size in chars
45	Set offset for writing	variable, offset
46	Write variable	char,char,...

Level 6 plasma panel task codes and parameters

Table 2

The routine PP-write cooperates with the interrupt service routine z80 int to put data on the screen. It first calls rsrv pnl to temporarily gain exclusive use of the panels to be written on. (If some other process has reserved one of the panels previously, the requesting process will block until the panel is released.) Pp write initiates an output transfer to send the command to the appropriate panel controllers. It then P's a semaphore for each panel accessed. As each panel completes the operation, the interrupt service routine vee's the appropriate semaphore. When all operations have completed, the requesting process will release the panels used by calling rls pnl and continue.

Touch Panel/Keyboard Input

When a character is typed on the keyboard, the controller sets bit 0 (the least significant bit) in its status byte and buffers the character. When the touch panel beams are interrupted, the controller sets bit 1 in the status byte and buffers the coordinates of the touch. If there is no operation currently in progress (i.e. no write to the panel or read from the keyboard or touch panel), the controller initiates an attention interrupt. When any interrupt is received (attention or otherwise), the interrupt service routine reads the controller status. This status is a word value with the status byte from the controller in the most significant byte, and flags from the MDC interface in the least significant byte. The interrupt routine checks bits 8 and 9 of the status word (bits 0 and 1 of the high order byte). If one of these is set, the routine initiates a read from the appropriate device and remembers that a read is in progress. When the read completes, the interrupt routine moves the data into either the keyboard or touch panel input buffer and signals the appropriate device process.

Note that all bits in this discussion are numbered in DEC format in which bit 0 is the least significant bit. This is the reverse of the way Honeywell numbers bits.

BIBLIOGRAPHY

1. "Assembly Language," July 1976
Honeywell Information Systems Inc., 200 Smith Street,
MS 486, Waltham, Mass. 02154
2. Brown, D.S.
1976, "Humanizing Data Management Systems: An Intelligent
Terminal Approach", Masters Thesis, CAC Document No. 186,
CCTC-Wad Document No. 6502.
3. "CINCPAC Study Report", 1976, CAC Document.
4. Dijkstra, E.J.
1968, "Co-operating Sequential Processes", Programming Languages,
F. Genuys, ed., Academic Press, New York, 1968.
5. Kopetzky, D.
Description of Remote Display Head Controller, to be
released.
6. "Honeywell Level 6 Minicomputer Handbook", August 1976,
Honeywell Information Systems Inc., 200 Smith Street,
MS 486, Waltham, Mass. 02154
7. "LSI-11 PDP-11/03 Processor Handbook", Digital Equipment
Corporation, Maynard, Massachusetts 01754.
8. "Operator's Guide," July 1976
Honeywell Information Systems Inc., 200 Smith Street,
MS 486, Waltham, Mass. 02154
9. "Program Development Tools," July 1976
Honeywell Information Systems Inc., 200 Smith Street,
MS 486, Waltham, Mass 02154
10. Ritchie, Dennis M.
"C Reference Manual", Bell Telephone Laboratories,
Murray Hill, New Jersey 07947.
11. Thompson, D., and Ritchie, D.M.
1975, "UNIX Programmer's Manual, Sixth Edition", Bell Telephone
Laboratories, Murray Mill, New Jersey 07974.
12. "Utility Programs," July 1976
Honeywell Information Systems Inc., 200 Smith Street,
MS 486, Waltham, Mass. 02154

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS
BEFORE COMPLETING FORM

1. REPORT NUMBER CAC Document #236; CCTC-WAD #7616		2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Research in Network Data Management and Resource Sharing INTELLIGENT TERMINAL PROGRAMMER'S MANUAL (Volume One)		5. TYPE OF REPORT & PERIOD COVERED Research	
		6. PERFORMING ORG. REPORT NUMBER CAC #236	
7. AUTHOR(s) Deborah S. Brown, Daniel J. Kopetzky, John R. Mullen, and David A. Willcox		8. CONTRACT OR GRANT NUMBER(s) DCA100-76-C-0088	
9. PERFORMING ORGANIZATION NAME AND ADDRESS Center For Advanced Computation University of Illinois, Urbana, Illinois		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS	
11. CONTROLLING OFFICE NAME AND ADDRESS Command and Control Technical Center Defense Communications Agency 11440 Isaac Newton Sq., Reston VA 22090		12. REPORT DATE October 31, 1977	
		13. NUMBER OF PAGES 169	
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED	
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE	
16. DISTRIBUTION STATEMENT (of this Report) No Restriction on Distribution			
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)			
18. SUPPLEMENTARY NOTES Copies of this report may be obtained from (11), above.			
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Intelligent Terminal			
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) The Programmer's Manual for the touch-oriented Intelligent Terminal developed for CCTC-WAD's Man-Machine Interface Project.			

5.10.84
EL 63e
no. 236

ENGINEERING LIBRARY *Engin*
UNIVERSITY OF ILLINOIS
URBANA, ILLINOIS

V. 2

CONFERENCE ROOM

Center for Advanced Computation



UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
URBANA, ILLINOIS 61801

CAC Document Number 236
CCTC-WAD Document Number 7616

Research in
Network Data Management and
Resource Sharing

**INTELLIGENT TERMINAL
PROGRAMMER'S MANUAL**

Volume Two of Two Volumes

October 31, 1977

The Library of the

MAY 24 1978

University of Illinois

The person charging this material is responsible for its return to the library from which it was withdrawn on or before the **Latest Date** stamped below.

Theft, mutilation, and underlining of books are reasons for disciplinary action and may result in dismissal from the University.

UNIVERSITY OF ILLINOIS LIBRARY AT URBANA-CHAMPAIGN

ENGINEERING

JUN 6 1978

MAR 14 1982

MAR 8 1982

CAC Document Number 236
CCTC-WAD Document Number 7516

Intelligent Terminal
Programmer's Manual

Volume Two of Two Volumes

Deborah S. Brown
Daniel J. Kopetzky
John R. Mullen
David A. Willcox

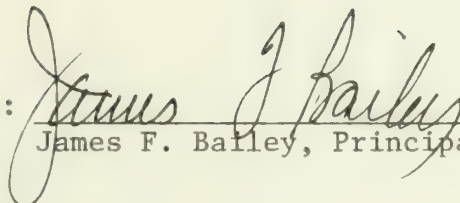
Prepared for the
Command and Control Technical Center
WWMCCS ADP Directorate
Defense Communication Agency
Washington, D.C.

under contract
DCA100-76-C-0088

Center for Advanced Computation
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

October 31, 1977

Approved for release:



James F. Bailey, Principal Investigator

APPENDIX A

IT Procedures by Functional Grouping

(For detailed descriptions of these routines see Appendix B.)

System Initialization Routines

‡ entry
 * fixup
 startup

I/O System Routines

clear_io
 close
 create
 delete
 flush
 io_init
 open
 peek
 read
 seek
 set_mode
 write

Device Drivers and Interrupt

Handlers

dk_driver
 kb_driver
 * kb_interrupt
 * peeper
 ph_driver
 phrint
 phxint
 tp_driver
 * tp_interrupt
 vip_proc
 ‡ vipint
 * viprint
 * vipxint
 ‡ z80_int

Touch Target Routines

scrunch
 tt_activate
 tt_arranger
 tt_cleanup
 tt_create
 tt_deactivate
 tt_delete
 tt_flash
 tt_label
 tt_lite
 tt_mark
 tt_move
 tt_outline
 tt_read
 tt_relabel
 tt_selections

Kernel Routines

alloc
 block
 * Bus_error
 creep
 deq
 enq
 enq_RQ
 error
 first_block
 free
 halt
 * int_disp
 kill
 ‡ levlp
 mfps
 mtps
 pause
 pee
 read_q
 suicide
 * Trap
 ‡ TRPHND
 vee
 write_q

String Manipulating Routines

cmp
 cvb
 get_token
 index
 ln_xpand
 parm_xpand
 verify

Plasma Panel Access Routines

area_lite
 erase
 ‡ get_pnl
 ‡ init_pnl
 ‡ pp_read
 ‡ pp_write
 put
 * putchar
 putdot
 putline
 ‡ rls_pnl
 ‡ rsrv_pnl
 screen_clear
 ‡ set_pnl
 ‡ Z80_LD

‡ These routines appear only in the Level 6 version of the IT system.

* These routines appear only in the LSI-11 version of the IT system.

Disk Access Routines

blk_alloc
 blk_free
 buf_io
 determine
 dir_open
 fclose
 fdelete
 fio
 fopen
 fseek
 ftrunc
 i_freemap
 init_drive
 init_fib
 pathname
 s_read
 s_write
 save_free
 search
 swab
 xopen
 zero_blk
 zero_sim

Data Display Routines

bar_graph
 digit_pad
 legend
 make_bar
 map
 refilter
 scale
 sh_map
 table

Communication Interface

‡ disable_io
 ‡ enable_io
 ‡ init_ccb
 ‡ ld_mlcp
 ‡ restart_io

Formatted Printing Routines

get_charset
 get_cursor
 get_env
 get_page_size
 get_pg
 get_size_chars
 ‡ ld_cs
 ‡ ld_page
 mk_page
 printf
 * put_ascii
 ‡ put_string
 ring_bell
 set_charset
 set_cursor
 set_env
 set_page
 str_num
 tiod
 tok_print

Other Routines

* cret
 * csv
 ‡ io
 ‡ iold
 kb_echo
 ldiv
 lrem
 mk_cursor
 pr_n_clear

‡ These routines appear only in the Level 6 version of the IT system.
 * These routines appear only in the LSI-11 version of the IT system.

APPENDIX B

Description of IT Procedures

(For a breakdown of these routines by function see Appendix A.)

NAME:

alloc

PURPOSE:

Allocates blocks of memory.

USAGE:

```
int *adr, *alloc(), size;
```

```
adr = alloc(size);
```

PARAMETERS:

size

An integer which is the number of words to allocate

RETURNS:

-1

If size contiguous words are not available.

Address of allocated memory otherwise.

CALLS:

Nothing

NAME:
 area_lite

PURPOSE:
 area_lite is used to erase or light a rectangular portion of the plasma panel.

USAGE:
 int x1, y1, x2, y2, mode;

 area_lite (x1, y1, x2, y2, mode);

PARAMETERS:

x1	integer	Left x coordinate of box
y1	integer	Lower y coordinate of box
x2	integer	Right x coordinate of box
y2	integer	Upper y coordinate of box
mode	integer	0 to erase, non-zero for light

RFTURNS:
 Nothing

CALLS:
 (LSI-11 version)
 put
 erase

 (Level 6 version)
 pp_write

NAME:

bar_graph

PURPOSE:

Display a data item as a bar graph on the plasma panel.

USAGE:

```
int x_orig, y_orig, width, height, num_values;
int values[...];
char *labels[...], *title;
```

```
bar_graph (x_orig, y_orig, width, height, num_values, labels,
           values, title);
```

PARAMETERS:

x_orig	Integer Left edge of the area to use for the graph, in characters
y_orig	Integer Bottom edge of the area to use for the graph, in characters
width	Integer Width of the area to use for the graph, in characters
height	Integer Height of the area to use for the graph, in characters
num_values	Integer Number of data values in this item
labels	Array of pointers to characters Pointers to the labels to use for the data elements
values	Array of integers The data values to display in the graph.
title	Pointer to character Points to the name to use for labeling the graph

RETURNS:

Nothing

CALLS:

make_bar	(bar graph)
printf	
put_ascii	
putline	
scale	(bar graph)
set_cursor	

NOTES:

This routine assumes characters are 8x16. It should be fixed to use the new variable character sizes.

If num_values is less than or equal to 0, this procedure will bomb.

Under the current scaling algorithm, the minimum data value is set to be relative 0, and all other values are decremented (or incremented) accordingly. If the minimum value is less than 0, this produces kind of funny results. It would be better if, when the minimum is negative, the axis was moved over and negative values were displayed as bars to the left of it.

NAME:

blk_alloc

PURPOSE:

Allocate blocks from disk drive freemap.

USAGE:

```
int    drv;  
int    block;
```

```
block = blk_alloc(drv);
```

PARAMETERS:

drv	integer	Drive number to be allocated from.
-----	---------	------------------------------------

RETURNS:

positive integer	Block number which has been allocated. (0 - 499)
-1	Error.

CALLS:

zero_block	Initializes a disk block to zeros.
------------	------------------------------------

NAME:
 blk_free

PURPOSE:
 Return a disk block to the freemap.

USAGE:

 int drv;
 int blk;
 int status;

 status = blk_free(drv, blk);

PARAMETERS:
 drv integer Disk drive number.
 blk integer Block number to be returned to the freemap.

RETURNS:
 -1 Block number out of range.
 0 No error.

CALLS:
 nothing

NAME:

block

PURPOSE:

Block is the process switcher. After saving R5 and R6 on the current process's stack, it removes a process from the ready queue, restores R5 and R6 for the new process from the base of its stack, and then returns to the new process.

USAGE:

block ();

PARAMETERS:

None

RETURNS:

Nothing

CALLS:

deq
error
suicide

NAME:

bufio

PURPOSE:

Perform buffered transfers of information to and from disk without knowledge of file structures or disk blocking.

USAGE:

```
struct  buffer *ev;  
int     blk,o,l,fcn;  
char    *buf;
```

```
status = bufio (ev, blk, o, buf, l, fcn);
```

PARAMETERS:

ev	pointer to buffer structure	Ev indicates the disk drive to use and supplies a one sector intermediate data buffer.
blk	integer	Disk block where data is to be found.
o	integer	Byte offset in the disk block where the transfer is to begin.
buf	pointer to character	A pointer to the user's buffer.
length	integer	Number of bytes to be transferred.
fcn	integer	IO function to perform, READ, WRITE, or FLUSH. These constants are defined in disksystem.incl to be 0, 1, and 2 respectively.

RETURNS:

0	No error.
-1	Error.

CALLS:

```
s_write  
s_read
```


NAME:

Bus_error

PURPOSE:

Handles bus errors on the LSI-11 IT. Any such errors encountered are considered fatal, and as such halt the machine.

CALLS:

error
printf

NOTES:

This routine is automatically invoked whenever the terminal encounters a bus error during execution. On the LSI, bus errors cause a trap thru location 4. The contents of location 4 are set to point to Bus_error by the low.o file, when the system is loaded. Changing the reference to Bus_error in the low file will result in having a different handler. Since bus errors are almost always errors, care should be taken in establishing the handler.

Bus_error should never be called from user routines. It is not callable from C routines.

NAME:

clear_io

PURPOSE:

clear_io removes all references to the calling process from the system i/o table. This requires closing any devices which have been opened by the process, plus removing the process from the list of processes waiting to open each device. The Level 6 version of this routine also makes sure that the plasma panel(s) are released.

USAGE:

clear_io ();

PARAMETERS:

None

RETURNS:

Nothing

CALLS:

(LSI-11 version)

close (io_sys)
deq (kernel)
enq (kernel)

(Level 6 version)

close (io_sys)
deq (kernel)
enq (kernel)
rls_pnl

NAME:

close

PURPOSE:

Close relinquishes ownership of a device. If there is a list of processes waiting for the device, close V's the first one on the list.

USAGE:

```
int close(), device_id, status;
```

```
if (close (device_id, &status) < 0) { error }
```

PARAMETERS:

device_id	An integer that uniquely identifies the device to close. This should be the value returned by open.
st_ptr	Pointer to an integer status word. The status word (not st_ptr) will be set to the status value returned by the device process.

RETURNS:

-1	If device_id is invalid, if the device is not owned by the calling process, or if the device process indicated catastrophic failure.
0	Normal return.
device status	The status value returned to close by the device process is returned indirectly to the caller. See the description of st_ptr under PARAMETERS.

CALLS:

write_q	(kernel)
pee	(kernel)
vee	(kernel)

NAME:

cmp

PURPOSE:

Compares two strings

USAGE:

```
int cmp(), length, relation;  
char *a, *b;
```

```
length = number_of_characters_to_compare;  
relation = cmp(a, b, length);
```

PARAMETERS:

a	pointer to character	Pointer to first string
b	pointer to character	Pointer to second string
length	integer	Number of characters to compare; usually the length of the strings.

RETURNS:

-1	if a < b
0	if a = b
1	if a > b

CALLS:

None

NOTES:

The comparison will terminate when either length characters have been checked, an inequality has been found, or a null is encountered in either string.

NAME:
create

PURPCSE:
Create a file on disk. If file already exists and is not a directory it is truncated to zero length.

USAGE:
int file_id, *status;
char *name;

file_id = create (name, stat_ptr)

PARAMETERS:
name pointer to character pointer to full name of file
stat_ptr pointer to integer status word to be returned.

RETURNS:
-1 Any error. (Unable to find directory, out of disk space, attempt to create an already existing directory...)
Positive integer File id, same as returned by open.

CALLS:
blk_alloc
close
ftrunc
open
read
save_free
seek
write

NAME:
creep

PURPOSE:
Creep is the procedure that creates processes for the IT operating system.

USAGE:
int creep(), stack_size, (*procedure)(), parm, priority, process_id;
process_id = creep (stack_size, procedure, parm, priority);

PARAMETERS:

stack_size	Integer number of words to make the new stack.
procedure	Pointer to the procedure that will be the process procedure.
parm	Integer. This is an arbitrary word that will be passed to the new process as a parameter. It is not used by creep.
priority	Integer used to specify the priority of the process. The bigger the number the higher the priority. System processes have the following priorities: device drivers -- 8 main (user process) -- 6 phone line peeper -- -1

RETURNS:
-1 if stack_size contiguous words of memory are not available.

An integer that is the ID of the process. In fact, this is the address of the base of the stack of the newly-created process.

CALLS:
alloc
enq_RQ

NAME:

cret

PURPOSE:

Restores registers on return from a C-language program.

USAGE:

This routine is jumped to automatically at the completion of a C-language program. It does not return to the routine that jumped to it, but rather to the routine that called that routine.

PARAMETERS:

None

RETURNS:

Nothing

CALLS:

None

NOTES:

This routine cannot be called explicitly by C-language programs.

This is a UNIX system routine. The object code for it is located in /lib/libc.a. Since it is a UNIX routine, the source for it is not included with the IT code.

NAME:

csv

PURPOSE:

Save registers upon entry to a C-language routine.

USAGE:

This routine is called automatically at the entry to a C-language program.

PARAMETERS:

None

RETURNS:

Nothing

CALLS:

None

NOTES:

This routine cannot be called explicitly from a C program.

This is a UNIX system routine. The object code for it is located in /lib/libc.a. Since it is a UNIX routine, the source for it is not included with the IT code.

NAME:
cvb

PURPOSE:
This routine converts a numeric character string to internal binary format. Input may be decimal or octal, the latter specified by a leading zero.

USAGE:

int cvb(), length, value;
char *string;

length = <length of string pointed to by 'string'>;
if (cvb (string, length, &value) < 0) { Conversion error }

PARAMETERS:

ptr	pointer to character	Pointer to string to be converted
length	integer	Length of string pointed to by ptr
value_ptr	pointer to integer	Place where value is returned

RETURNS:
0 if everything is OK
-1 if a non-numeric character was encountered

The word pointed to by value_ptr is changed. See PARAMETERS.

CALLS:
None

NAME:
delete

PURPOSE:
Remove a file from a floppy disk.

USAGE:
char *filename;
int status, *st_ptr;

status = delete (filename, st_ptr);

PARAMETERS:
filename pointer to a character string indicating
the file to be deleted.
st_ptr pointer to an integer status word.

RETURNS:
-1 File not found or an error occurred in the delete operation.
0 No error.

CALLS:
open
pee
write_q

NAME:

determine

PURPOSE:

Converts a file or device name into an index into the device table (DEV_TAB). If the name matches one of the fixed device names in DEV_TAB, the corresponding index is returned. Otherwise the name is treated as a file name. An attempt is made to open the file. If successful determine will return the DEV_TAB slot by which the file may be referenced.

USAGE:

```
char    *name;
int      index;

index = determine (name);
```

PARAMETERS:

name	pointer to character	Name of device or file.
------	----------------------	-------------------------

RETURNS:

positive integer	Index into DEV_TAB.
-1	Error, file or permanent device not found.

CALLS:

fopen
init_drive
pathname

NAME:

deq

PURPOSE:

deq is a general dequeue routine. It will remove the next item from a queue that is in the standard format. An attempt to dequeue from an empty queue is disastrous.

USAGE:

```
int q_entry;  
struct queue_head que;  
  
q_entry = deq (&que);
```

PARAMETERS:

q_ptr A pointer to the header of a standard queue.

RETURNS:

The single-word value of the oldest entry on the queue.

CALLS:

mtps	(kernel)
mfps	(kernel)
error	(kernel)

NOTES:

Deq makes itself non-interruptable for most of its operation.

An attempt to deq from an empty queue results in a call to error. The current version of error does not return, and deq is written with that knowledge in mind. If error is ever modified to return, deq will also have to be altered.

NAME:
digit_pad

PURPOSE:
Display a ten key numeric pad and some control keys. Allows the user to input numeric data by touching the touch panel.

USAGE:
int digit_pad(), touch, top_of_pad, status, entered_num;

entered_num = digit_pad (touch, top_of_pad, &status);

PARAMETERS:

touch	Integer File id of the touch panel, returned by open
top_of_pad	Integer Coordinate of touch grid which is to be the top of the numeric pad
rtn_status	Pointer to integer Points to a word which indicates the status on return. The word is set to -1 if the user hit Cancel, otherwise it is set to 0.

RETURNS:
num Integer The number entered by the user.
(Returns values indirectly through parameters.)

CALLS:

activate	(old touch target routine)
add_command	(old touch target routine)
del_command	(old touch target routine)
get_command	(old touch target routine)
get_cursor	
printf	
set_cursor	

NOTES:

This procedure is specifically related to the Janus demo system. It is included here primarily for completeness since it is referenced by map. It may be of some use as a guideline if someone writes a general data-input-via-touch routine sometime in the future.

This routine was written before the print and touch target routines were updated. It will not run with the new versions of touch target handlers. It needs to be rewritten to use the new routines.

A lot of the funkyness with labeling in this procedure is done because calling the old target routines moved the cursor. With the new print routines this should all get much easier.

NAME:

dir_open

PURPOSE:

Open a file as if it were the directory.

USAGE:

```
struct  buffer      *ev;  
struct  file_info  *fib;  
int      index_b;  
int      status;  
  
status = dir_open (ev, fib, index_b);
```

PARAMETERS:

ev	pointer to a buffer structure	Disk data buffer area to use (also defines the drive number in use).
fib	pointer to file_info structure	File_info structure to be filled in so other routines can access the directory.
index_b	integer	Index block of the file to be used as a directory.

RETURNS:

-1	Error.
0	No error.

CALLS:

```
fio  
init_fib  
xopen
```


NAME:

disable_io

PURPOSE:

Turn off input or output for a specified MLCP-type device.

USAGE:

int chan;

if (disable_io (chan) != 0) { error }

PARAMETERS:

chan Integer Id of channel to be turned off.

RETURNS:

-1 If io instructions failed.

0 If io successfully halted.

CALLS:

io

NAME:

dk_driver

PURPOSE:

The disk process allows application programs to read from and write to the floppy disk.

USAGE:

```
extern struct queue_head DSK_Q;  
struct req_block rb;  
  
write_o(&DSK_Q, &rb);
```

PARAMETERS:

None in the usual sense. Communication is via the input queue. Items on the input queue are pointers to requests blocks.

RETURNS:

Never returns. When an application program request has been handled, the semaphore associated with the request is vee'd.

CALLS:

```
fio  
bufio  
fseek  
fclose  
ftrunc
```

NOTES:

This routine should never be called directly by user programs. Access should be through I/O routines read, write, open, etc.

This process has a priority of 8.

A read from the disk will return a string containing the minimum of the number of bytes requested and the number of unread bytes in the specified file. The returned string is not null-terminated.

NAME:
 enable_io

PURPOSE:
 Allow input or output from an MLCP-type device.

USAGE:
 int chan, dev;

 if (enable_io (chan, dev) != 0) { error }

PARAMETERS:

chan	Integer Id of channel to be started up. If this number is even, it will be taken to be an input channel; otherwise it will be assumed to be output.
dev	Integer IT id of device containing this channel. Used as an index into the list of CCP starting addresses which is ordered by IT id.

RETURNS:

-1	If any of the io instructions failed
0	If io enabled

CALLS:
 io

NAME:

enq

PURPOSE:

Enq is a general purpose enqueue routine. It is used to add an element to a circularly linked list of items.

USAGE:

```
int value;
struct queue_head que;

enq (&que, value);
```

PARAMETERS:

q_ptr	A pointer to the head of a standard queue.
value	An arbitrary one-word value that will be enqueued.

RETURNS:

Nothing

CALLS:

mtps	(kernel)
mfps	(kernel)
error	(kernel)

NOTES:

Enq makes itself non-interruptable for most of its execution.

Enq calls error if there are no free queue elements. The current version of error does not return, and enq makes use of that fact. If error is ever changed to allow returns, enq will have to be modified to account for that.

NAME:

enq_RQ

PURPOSE:

Enqueue an item on the system ready queue. This differs from a normal enq in that the ready queue is an ordered list rather than a strictly FIFO queue.

USAGE:

```
int value, priority;
```

```
enq_RQ (value, priority);
```

PARAMETERS:

value	An arbitrary single-word value. Normally, this is the ID of the process which is being added to the ready queue.
priority	An integer specifying the priority of the new item. The greater this value, the higher the priority of the item.

RETURNS:

Nothing

CALLS:

error	(kernel)
mfps	(kernel)
mtps	(kernel)

NOTES:

enq_RQ makes itself non-interruptable for most of its operation.

If there are no free queue elements, enq_RQ calls error. The current version of error does not return, and enq_RQ knows this. If error is modified to return, enq_RQ will have to be modified.

NAME:

entry

PURPOSE:

This routine performs some initialization required for starting up a system on the Level 6. It sets up the stack pointers and finds the high end of memory that is available for use by the IT system.

USAGE:

This routine is entered only at system startup.

PARAMETERS:

On entry, B3 points to the high end of memory.

RETURNS:

Never - the call to startup never returns.

CALLS:

startup

NOTES:

This routine is not a procedure in the normal sense. It exists as a section of code in the object module LOW.A.

NAME:

erase

PURPOSE:

Erase is used to perform parallel accesses to the plasma panel. It can selectively write each dot in a group of 16 dots arranged vertically on the panel.

USAGE:

```
int X, Y, scalar, vector[..], count;
```

```
erase (X, Y, scalar, count, 0);
```

or

```
erase (X, Y, vector, count, 1);
```

PARAMETERS:

X	An integer that is the X coordinate of the first vector to erase. All subsequent vectors will be erased at succeeding X coordinates.
Y	An integer that is the Y coordinate of the lower end of the 16 dot vector.
vector	This is either an integer or a pointer to a group of integers. In either case, the bits in the integer(s) which are set will cause the corresponding dot to be lit. The low order bit of the integer corresponds to the lowest dot in the vector.
count	An integer that tells how many vectors to erase on the panel. If flag is 0, one vector will be erased count times; otherwise, count vectors will be erased once each.
flag	An integer that tells how to interpret vector, above. If it is 0, vector is considered an integer and is erased out count times. If flag is nonzero, vector is treated as a pointer to an array of integers that is count long.

RETURNS:

Nothing

CALLS:

(LSI-11 version)

Nothing

(Level 6 version)

pp_write

rls_pnl

rsrv_pnl

NOTES:

The bus address (in the LSI version) or the channel address (in the Level 6 version) of the plasma panel interface is "known" by this routine. If the address ever changes, the address defines will have to be changed and erase will have to be recompiled.

NAME:
error

PURPOSE:
Convert an error code into a message and print the message.

USAGE:
int err_no;

error (err_no);

PARAMETERS:
err_no An integer error code. This should be one of the standard codes defined in constants.incl.

RETURNS:
Nothing Error DOES NOT return to its caller.

CALLS:
(LSI-11 version)
halt
mfps
mk_page
mtps
printf
set_page

(Level 6 version)
halt
io
mfps
mk_page
mtps
printf
set_page

NOTES:
No checking is done to ensure that the error code is within range.
A bad error code will result in a garbage message.

Error does not return to its caller. It should probably be fixed to know the difference between fatal and non-fatal errors.

The Level 6 version of this routine snatches the plasma panel away from which ever process is currently using it. Specifically, it sets the global variable pp_owner equal to the process id of the current process. It also turns off interrupts from the remote display head controllers.

NAME:

fclose

PURPOSE:

Close an open file, updating directory if necessary.

USAGE:

```
struct  buffer      *ev;  
struct  file_info   *fib;  
int      status;
```

```
status = fclose (ev, fib);
```

PARAMETERS:

```
ev      pointer to buffer structure  
        Indicates drive number and where a  
        temporary buffer can  
        be found.  
fib     pointer to file_info structure  
        The file_info structure of the file being closed.
```

RETURNS:

```
0          No error.  
-1         Error.
```

CALLS:

```
fio  
bufio  
init_fib
```


NAME:

fdelete

PURPOSE:

Removes an open file and all its blocks, returning space to the freemap.

USAGE:

```
struct  buffer      *ev;  
struct  file_info   *fib;  
int      status;  
  
status = fdelete (ev, fib);
```

PARAMETERS:

ev	pointer to buffer structure	Indicates where an intermediate buffer can be found for accessing the disk.
fib	pointer to file_info structure	File_info structure for the file to be deleted.

RETURNS:

0	No error.
-1	Error.

CALLS:

```
blk_free  
bufio  
dir_open  
fio  
fseek  
ftrunc  
i_freemap  
save_free
```


NAME:

fio

PURPOSE:

Perform file level input/output without respect to IT device specifics. Handles holes, extension, multiblock & overlapping block operations.

USAGE:

```
struct  buffer      *ev;  
struct  file_info   *fib;  
char    *userbuf;  
int      l, fcn, status;  
  
status = fio (ev, fib, userbuf, l, fcn);
```

PARAMETERS:

ev	pointer to buffer structure	Data buffer area to be used in accessing the disk.
fib	pointer to file_info structure	File_info structure which defines the disk file to be used in obtaining data.
userbuf	pointer to character	Indicates where the data will be transferred to.
l	integer	length of transfer desired.
fcn	integer	IO function code, READ, WRITE or FLUSH. These are defined in disksystem.incl to be 0, 1, and 2 respectively.

RETURNS:

-1	Any error while performing requested operation.
integer	Actual number of bytes transferred.

CALLS:

blk_alloc
bufio
save_free
zero_sim

NAME:
 first_block

PURPOSE:
 First_block is a alternate entry point into block. It is used as the first call to block from startup after the system has been started. It selects a process from the ready queue and starts it executing.

USAGE:
 first_block();

PARAMETERS:
 None

RETURNS:
 Never returns

CALLS:
 deq
 error
 suicide

NOTES:
 Should only be called by startup.

NAME:

fixup

PURPOSE:

Fixup is used to determine the amount of memory that is available to the LSI-11 IT, and to relocate the stack pointer to point to the top end of memory. It maintains the relationship between R5 and R6 so it may be called by a C program.

USAGE:

```
int fixup (), reserve, size;
```

```
size = fixup (reserve);
```

PARAMETERS:

reserve	integer	Amount of space to leave untouched at the top of memory. This is to leave a piece of inviolate memory that can be used for bootloaders, patches, or anything else which shouldn't be destroyed by initialization.
---------	---------	---

RETURNS:

The highest usable address in the machine.

CALLS:

None

NOTES:

The values of any local variables will be destroyed by this call. Therefore, this call must be the first assignment statement.

NAME:

flush

PURPOSE:

Flush causes the process associated with a device to discard any buffered input and output. It can be successfully invoked only by the process that currently "owns" the device.

USAGE:

```
int flush(), device_id, status, length;
```

```
length = flush (device_id, &status);  
if (length < 0) { error }
```

PARAMETERS:

device_id	An integer that uniquely identifies the device to be flushed. This should be the value returned by a call to open.
st_ptr	Pointer to an integer. The integer (not st_ptr) will be set equal to the status returned by the device process.

RETURNS:

-1	If the device does not exist, if the calling process does not own the device or if the device process indicates a catastrophic failure.
data length	Under all other circumstances, the data length field of the request block is returned. Most device processes set this value to the number of bytes that were flushed.
device status	Returned indirectly via st_ptr. See the description of st_ptr under PARAMETERS.

CALLS:

write_q	(kernel)
pee	(kernel)

NAME:

fopen

PURPOSE:

Given a file name, disk drive number and current directory index block, search for and open the file.

USAGE:

```
char          *name;  
struct file_info *fib;  
int           drv, root, status;
```

```
status = fopen (name, fib, drv, root);
```

PARAMETERS:

name	pointer to character	Represents a character string for the name of the file.
fib	pointer to file_info structure	If the file is found, this file_info structure will be initialized to allow the file to be accessed by other routines.
drv	integer	Drive where directory is to be found.
root	integer	The index block of the directory to where the file is to be found.

RETURNS:

-1	any failure down the line
0	success

CALLS:

```
dir_open  
fio  
pathname  
xopen
```


NAME:

free

PURPOSE:

Return memory to the pool of free memory.

USAGE:

int size, *aa;

free (size, aa);

PARAMETERS:

size Integer number of words to return.

aa Address of memory to be freed.

RETURNS:

Nothing

CALLS:

Nothing

NOTES:

The parameter 'aa' is usually the address that was earlier returned by alloc.

NAME:

fseek

PURPOSE:

Change the position of read/write pointer for a disk file.

USAGE:

```
struct file_info    *fib;  
int                 off, type, status;
```

```
status = fseek (fib, off, type);
```

PARAMETERS:

fib	pointer to file_info structure	Indicates the file whose read/write pointer is to be changed.
off	integer	Offset
type	integer	Type of seek. (see the "seek" command).

RETURNS:

0	No error.
-1	Error.

CALLS:

nothing

NAME:
ftrunc

PURPOSE:
Free space used by a file. Unlinks blocks from the index block.

USAGE:
struct buffer *ev;
struct disk_info *fib;

status = ftrunc (ev, fib);

PARAMETERS:
ev pointer to buffer structure
Indicates a buffer through which the disk
may be accessed.
fib pointer to file_info structure
File_info structure for the file to
be truncated.

RETURNS:
-1 Error.
0 No error.

CALLS:
blk_free
bufio
fio
zero_blk

NAME:

get_charset

PURPOSE:

Returns the id of the current character set.

USAGE:

struct cs_desc *get_charset(), *CS;

CS = get_charset();

PARAMETERS:

None

RETURNS:

Pointer to the charset currently being used.

CALLS:

Nothing

NOTES:

Complementary function to set_charset.

NAME:

get_cursor

PURPOSE:

Fills two variables with the current position of the cursor in character coordinates.

USAGE:

```
int x, y;
```

```
get_cursor (&x, &y);
```

PARAMETERS:

x_ptr =	Pointer to integer Points to variable which is to be set to the x coordinate value.
y_ptr =	Pointer to integer Points to variable which is to be set to the y coordinate value.

RETURNS:

Nothing

(Returns values indirectly through the parameters.)

CALLS:

(LSI-11 version)

Nothing

(Level 6 version)

ld_page

pp_read

NOTES:

Complimentary procedure to set_cursor.

NAME:

`get_env`

PURPOSE:

Fills an `env_desc` structure with the values specifying the currently used charset, page, and cursor position. In the Level 6 version, the list of selected panels is also saved.

USAGE:

```
struct env_desc environment;  
  
get_env (&environment);
```

PARAMETERS:

<code>env_ptr =</code>	Pointer to <code>env_desc</code> structure
	Pointer to the structure which is to be filled in.

RETURNS:

Nothing
(Returns values indirectly through the parameter.)

CALLS:

(LSI-11 version)

```
get_charset  
get_cursor  
get_pg
```

(Level 6 version)

```
get_charset  
get_cursor  
get_pg  
get_pnl
```

NOTES:

Designed to save the current printing environment, usually so it can be restored later via `set_env`.

NAME:

get_page_size

PURPOSE:

Returns the size of the current page in character dimensions.

USAGE:

```
int page_width, page_height;  
  
get_page_size(&page_width, &page_height);
```

PARAMETERS:

w_ptr	Pointer to integer Pointer to the variable to be filled in with the width of the page.
h_ptr	Pointer to integer Pointer to the variable to be filled in with the height of the page.

RETURNS:

Nothing
(Returns values indirectly through the parameters.)

CALLS:

Nothing

NAME:

`get_pg`

PURPOSE:

Fills in a `page_desc` structure with the description of the current page.

USAGE:

```
struct page_desc pg;
```

```
get_pg (&pg);
```

PARAMETERS:

<code>pg_ptr</code>	Pointer to <code>page_desc</code> structure
	Pointer to the structure to be filled in.

RETURNS:

Nothing
(Returns values indirectly through the parameters.)

CALLS:

Nothing

NOTES:

Complimentary function to `set_page`.

NAME:

`get_pnl`

PURPOSE:

Used to find out which panel(s) a call to the panel access routines will currently effect.

USAGE:

`int ppu;``get_pnl (&ppu);`

PARAMETERS:

`ppu_p` pointer to integer

Address where a word specifying which panels are currently accessed should be stored.

If this is 1, then future panel operations will go to panel 0. A value of 2 specifies panel 1 and 4 specifies panel 2. These values are additive, so a ppu value of 5 (for instance) means future operations will effect panels 0 and 2.

RETURNS:

Returns values thru parameters.

CALLS:

None

NAME:
 get_size_chars

PURPOSE:
 Returns the size of a character in the current charset (in dots).

USAGE:
 int char_width, char_height;

 get_size_chars(&char_width, &char_height);

PARAMETERS:

w_ptr	Pointer to integer Pointer to variable to be set to the width of a character, in dots.
h_ptr	Pointer to integer Pointer to variable to be set to the height of a character, in dots.

RETURNS:
 Nothing
 (Returns values indirectly through the parameters.)

CALLS:
 Nothing

NAME:

get_token

PURPOSE:

Parses the next token from a string. The returned token is maximal and contains no delimiters.

USAGE:

```
int get_token(), token_len;  
char *buf_ptr, token[...], delimiters[...];  
  
token_len = get_token(buf_ptr, token, delimiters);
```

PARAMETERS:

buf_ptr	pointer to character	Pointer to the source string. The string must be null terminated.
tok_ptr	pointer to character	Pointer to where token should be put. The resulting string will be null terminated.
delim_ptr	pointer to character	Pointer to string of delimiters. The string must be null terminated.

RETURNS:

integer - Number of characters scanned, or zero if no non-delimiters were found. Note that this may be greater than the length of the actual token.

CALLS:

verify

NOTES:

There is no checking for token length. It is up to the caller to make sure that the buffer is large enough to hold the returned token.

NAME:

halt

PURPOSE:

Used to halt the processor immediately. The processor is stopped by a halt instruction, so no further processing is done by any process.

USAGE:

halt ();

PARAMETERS:

None

RETURNS:

This routine never returns to its caller.

CALLS:

None

NAME:

i_freemap

PURPOSE:

Initializes the freemap for a drive.

USAGE:

```
int    drv;  
int    status;
```

```
status = i_freemap (drv);
```

PARAMETERS:

```
drv      integer      Drive number.
```

RETURNS:

```
-1      Error.  
0       No error.
```

CALLS:

bufio

NAME:

index

PURPOSE:

This is the PL/1 index function. It returns the starting position of the first occurrence of the second string within the first string.

USAGE:

```
int index(), position;  
char in_str[...], of_str[...];  
  
position = index(in_str, of_str);
```

PARAMETERS:

in_str	pointer to character	Pointer to string to be searched in
of_str	pointer to character	Pointer to string to be searched for

RETURNS:

int - Index of second string in first, or -1 if second string does not occur in first.

CALLS:

Nothing

NAME:

init_ccb

PURPOSE:

Initialize an MLCP CCB to use a specified buffer.

USAGE:

```
int chan, buf_size, cntrl;  
char buf[];
```

```
if (init_ccb (chan, buf, buf_size, cntrl) != 0) { error }
```

PARAMETERS:

chan	Integer Channel id of the channel (half of line) whose CCB is to be initialized.
buf	Pointer to character The buffer the CCB should use.
buf_size	Integer Size of the buffer.
cntrl	Integer Control word to be sent to the CCB.

RETURNS:

-1	If an error in the io was encountered.
0	Otherwise

CALLS:

```
io  
iold
```


NAME:
init_drive

PURPOSE:
Read the first directory entry in the root file of a disk
and initialize the memory resident freemap.

USAGE:
int drv, status;

status = init_drive (drv);

PARAMETERS:
drv integer Drive number.

RETURNS:
-1 Error.
0 No error.

CALLS:
dir_open
fio
fseek
i_freemap

NAME:

init_fib

PURPOSE:

Force suitable parameters into the fib so that file level reads/writes can be done.

USAGE:

```
struct file_info      *fib;
int                   index_b, off, status;

status = init_fib (fib, index_b, off);
```

PARAMETERS:

fib	pointer to file_info structure	The file_info structure to be initialized.
index_b	integer	Index block of the file.
off	integer	Initial offset in the file.

RETURNS:

-1	Index block out of range.
0	No error.

CALLS:

nothing

NAME:

init_pnl

PURPOSE:

Performs the initialization necessary to start up the z80 display head controller.

USAGE:

init_pnl ();

PARAMETERS:

None

RETURNS:

Nothing

CALLS:

free
get_pnl
pp_write
set_pnl
Z80_LD

NOTES:

This routine is defined only for the Level 6 version of the IT.

This routine is normally called only at system initialization. There should be no dire effects if it is called at some other time. However, the z80 won't actually be re-loaded. The character set and page descriptors will be reset.

Init_pnl does a "free" on the in-memory copy of the z80 microcode after it has been written out. This allows the 3-odd K of memory to be used for other purposes. It does mean, however, that the z80s can never be re-loaded without re-loading the IT system from scratch.

NAME:

int_disp

PURPOSE:

This is the interrupt handler for all valid interrupts. It saves the registers on the system interrupt stack and selects which routine will handle the interrupt.

USAGE:

Entered by an interrupt, only.

PARAMETERS:

None

RETURNS:

Nothing

CALLS:

kb_interrupt
phrint
tp_interrupt
phxint
viprint
vipxint

NAME:

io_init

PURPOSE:

Io_init is used to initialize the I/O tables for the IT.

USAGE:

io_init()

PARAMETERS:

None

RETURNS:

Nothing

CALLS:

Nothing

NAME:

io

PURPOSE:

Performs the Level 6 IO instruction. Allows C programs to read or write control information to the device controllers.

USAGE:

```
int channel, code, data;
```

```
io (channel, code, &data);
```

PARAMETERS:

channel	integer	The channel address of the device
code	integer	A function code for the type of operation to perform. These codes are described in the Level 6 Minicomputer Handbook.
data_p	pointer to integer	This is either a pointer to the data to be written to the device controller, or the address where a word returned by the controller can be stored.

RETURNS:

0	if io instruction was accepted
-1	if instruction was not accepted. This ususally means that the device is busy.

CALLS:

None

NAME:

iold

PURPOSE:

Performs the Level 6 IOLD instruction. Allows C programs to specify to a device controller the address and length of data to be transferred in an io operation.

USAGE:

```
int channel, count;  
char buffer[];
```

```
iold (channel, buffer, count);
```

PARAMETERS:

channel	integer	The channel address of the device
buffer	pointer to character	Address of the buffer for reading or writing.
count	integer	Count of characters to transfer

RETURNS:

0	if the iold instruction was accepted.
-1	if the instruction was not accepted. This usually means that the device is busy.

CALLS:

None

NAME:
 kb_driver

PURPOSE:
 The keyboard process allows application programs to read from the keyboard.

Kb_driver works in conjunction with the keyboard interrupt handler, kb_int (on the LSI-11), or the Z80 interrupt handler, z80_int (on the Level 6). It accepts data from the interrupt handler and buffers it until requested by an application program.

USAGE: (See NOTES)
 external struct queue_head KBP_Q;
 struct req_block rb;
 int special_value;

 write_q (&KBP_Q, &rb);
 or
 write_q (&KBP_Q, special_value); /* special_value must be odd in the
 LSI11 version, and less than 256 in
 the Level 6 version. */

PARAMETERS:
 None in the usual sense. Communication is via the input queue. Items on the input queue can be pointers to request blocks, characters from the keyboard interrupt handler (on the LSI-11), or signals from the Z80 interrupt handler that there is data in the input buffer (on the Level 6).

RETURNS:
 Never returns. When application program requests have been handled, their semaphores are vee'd.

CALLS:
 (LSI-11 version)
 get_size_chars
 put_ascii
 read_q
 ring_bell
 vee

 (Level 6 version)
 get_size_chars
 printf
 read_q
 ring_bell
 vee

NOTES:
 This routine should never be called directly by user programs. Access should be thru I/O routines read, peek, etc.

 This process has a priority of 8.

 A read from the keyboard will return a string containing the minimum of

the number of characters requested and the number of characters in the keyboard buffer. If the buffer is empty, the next character from the keyboard is returned.

If a second read comes in before the first has been vee'd, the first will be lost. This is defined to be an impossible situation if the read subroutine is used, but may occur during direct communication of user programs with kb_driver.

NAME:

kb_echo

PURPOSE:

Turns echoing of characters by the keyboard driver on or off.
When turning echoing on, will set the keyboard drivers's internal cursor position to be the same as the current printing position.

USAGE:

```
int      kb_id;

kb_echo (kb_id, 0);
      or
kb_echo (kb_id, 1);
```

PARAMETERS:

kb	Integer	Id of opened keyboard.
mode	Integer	1 to start echoing, 0 to stop.

RETURNS:

Nothing

CALLS:

```
get_cursor
set_mode
```

NOTES:

This procedure is not part of the IT system software. As such it may not be revised to be compatible with future versions of the system. The object code for this procedure is not contained in the system library libI.a

NAME:

kb_interrupt

PURPOSE:

This is the interrupt handler for the keyboard. It forwards characters from the keyboard to the keyboard process, which eventually gives them to the application programs.

USAGE:

Called as an interrupt handler, only.

PARAMETERS:

None

RETURNS:

Nothing

CALLS:

write_q (kernel)

NAME:

kill

PURPOSE:

Kill will force a process to commit suicide.

USAGE:

```
int kill(), proc_id;
```

```
if (kill(proc_id) < 0) { error }
```

PARAMETERS:

proc_id	The ID of the process to kill
---------	-------------------------------

RETURNS:

-1	If the stack of the target process is garbled.
0	Otherwise

CALLS:

Nothing

NOTES:

Any process can kill any other process. There is no "parent-offspring" relationship between processes.

NAME:

ld_cs

PURPOSE:

This routine performs the processing required to ensure that the "current" charset on the Level 6 is the same as that on the z80.

USAGE:

ld_cs ();

PARAMETERS:

None

RETURNS:

Nothing

CALLS:

get_pnl
pp_write
rls_pnl
rsrv_pnl
set_pnl

NAME:

ld_mlcp

PURPOSE:

Loads and initializes the multiline communications processor (MLCP).

USAGE:

int ld_mlcp ();

if (ld_mlcp ()) { error }

PARAMETERS:

None

RETURNS:

-1 if loading the MLCP failed

0 if loading succeeded

CALLS:

ZQMLIN (Honeywell-supplied support routine)

NAME:

ld_page

PURPOSE:

Loads the current Level 6 page descriptor into the z80 panel controller.

USAGE:

ld_page ();

PARAMETERS:

None

RETURNS:

Nothing

CALLS:

get_pnl
pp_write
rls_pnl
rsrv_pnl
set_pnl

NOTES:

The page descriptor sent to the z80 consists of eight words. They are, in order:

- 0) The left edge of the page,
- 1) The width of the page,
- 2) The bottom of the page,
- 3) The height of the page,
- 4) The width of the side page margin - i.e. how far in from the side edges of the page should printing start,
- 5) The width of the top and bottom margin of the page,
- 6) Flags which control printing on the page, (see the write-up for the z80 panel controller for descriptions of these flags) and,
- 7) The width of the border to be drawn around the page - this is zero for now.

NAME:

ldiv

PURPOSE:

Performs unsigned division.

USAGE:

```
int quotient, divisor, hi_dividend, lo_dividend;
```

```
quotient = ldiv (hi_dividend, lo_dividend, divisor);
```

PARAMETERS:

hi_dividend	Integer	Higher 16 bits of the 32-bit number to be divided. This parameter plus the next one are taken together and are interpreted as an unsigned 32-bit number.
lo_dividend	Integer	Lower 16 bits of the dividend.
divisor	Integer	Number to be divided into the dividend. This is treated as a standard 2's complement integer.

RETURNS:

quotient	Integer	Result of integer division performed on the parameters.
----------	---------	---

CALLS:

Nothing

NAME:

legend

PURPOSE:

Draws and shades one box of the legend for the shaded map. Positions the cursor to the right of the box for labeling.

USAGE:

int x, y, shades;

legend (x, y, shades);

PARAMETERS:

x	Integer	Coordinate of the left edge of the box, in characters
y	Integer	Coordinate of the bottom edge of the box, in characters
shade	Integer	Number of the shade to use

RETURNS:

Nothing

CALLS:

put
putline
set_cursor

NOTES:

This uses 8x16 dot characters hard coded in. This should be modified to use the new variable character sizes in the new print routines.

NAME:

levlp

PURPOSE:

This routine controls access to the device interrupt service routines in the Level 6.

USAGE:

This routine is entered whenever there is an interrupt.

PARAMETERS:

None

RETURNS:

Nothing

On return from an interrupt routine, levlp does an "inhibit" LEV instruction. This causes the interrupt routine to be deactivated, and normal processing is resumed.

CALLS:

A device interrupt routine. The interrupt save areas are set up so that when levlp is entered, the address of the entry point of the appropriate interrupt routine is contained in register B3.

NOTES:

This routine does not exist as a normal procedure. It exists as a section of code in the object module LOW.A.

levlp passes the address of the interrupting device to the interrupt routine as a parameter. This allows one interrupt routine to handle more than one of the same kind of device.

NAME:

`ln_xpand`

PURPOSE:

To expand a string, performing formatted parameter replacement similar to printf. The returned string is null terminated.

USAGE:

```
int ln_xpand(), length;  
char out[...], *in;
```

```
length = ln_xpand(out, in, parm1, parm2,...);
```

PARAMETERS:

out Pointer to character

Points to the array to be filled with the expanded string

in Pointer to character

Points to the format string, which is the same as a printf format string.

parms <any parameter to be expanded>

This is the first of the parameters to be expanded and inserted in the output string. There can be an any number of such parameters. The type and meaning of each such parameter will be interpreted according to the type of format by which it is to be expanded.
(Note that floating point parameters are not handled at this time.)

RETURNS:

Integer

The length of the expanded string including the trailing null

(Returns expanded string indirectly through parameter out.)

CALLS:

`parm_xpand`

NOTES:

No check is made to avoid overrunning the output buffer.

Format specifications must be in the simpler IT printf style. This means no width or precision specifications. This is due to simplifying assumptions made in `parm_xpand`. See the description of printf for a list of the formats available.

The flag character can be changed by changing the define flag.

NAME:

lrem

PURPOSE:

Calculate remainder for unsigned division.

USAGE:

```
int remainder, divisor, hi_dividend, lo_dividend;
```

```
remainder = lrem (hi_dividend, lo_dividend, divisor);
```

PARAMETERS:

hi_dividend	Integer Higher 16 bits of the 32-bit number to be divided. This parameter plus the next one are taken together and are interpreted as an unsigned 32-bit number.
lo_dividend	Integer Lower 16 bits of the dividend.
divisor	Integer Number to be divided into the dividend. This is treated as a standard 2's complement integer.

RETURNS:

remainder	Integer Remainder of integer division performed on the parameters.
-----------	--

CALLS:

Nothing

NAME:

make_bar

PURPOSE:

Prints one bar of specified width on the plasma panel.

USAGE:

int x, size, y;

make_bar (x, size, y);

PARAMETERS:

x	Integer	Left edge of the bar, in dots
size	Integer	Width of the bar, in dots
y	Integer	Coordinate of the bottom edge of the bar, in character lines

RETURNS:

Nothing

CALLS:

put

NOTES:

This uses 8x16 characters. It should be modified to use the new variable character sizes.

NAME:

map

PURPOSE:

Display one data item as a shaded map along with several touch targets. The targets allow the user to specify that the data should be replotted as a bar chart or table, to inspect the data for one county, to change the data for one county, or to proceed. This procedure interprets the user's touches and will perform the actions of allowing the user to inspect or change data.

USAGE:

```
int map(), touch, num_shades, values[...], num_areas;  
char *map_label;  
int touched;
```

```
touched = map(touch, num_shades, values, num_areas, map_label);
```

PARAMETERS:

touch	Integer File id of touch panel, returned by open
num_shades	Integer Number of different shades to use on the map
values	Array of integers Values to be associated with the counties. These may be changed by the user.
num_areas	Integer Number of counties
map_label	Pointer to character Name of data item, used to label map

RETURNS:

num	Integer	Returns the value of the button that was touched by the user. This value is used by the calling routine to determine what to do next.
-----	---------	---

(Returns values indirectly through the parameters.)

CALLS:

activate	(old touch target routine)
add_command	(old touch target routine)
area_lite	
cnty_buttons	(map routine - not documented)
deactivate	(old touch target routine)
del_command	(old touch target routine)
digit_pad	
get_command	(old touch target routine)
printf	
putline	
set_cursor	
sh_map	(map routine)
shade_county	(map routine - not documented)

NOTES:

This procedure is specifically related to the Janus demo system. It is included here primarily for completeness. The only generally useful part of it is the section at the beginning that draws the outline of the map.

The map description itself is completely external to all the map routines. (It is stored in the data structures polys and tmplt.) However the map procedure makes several assumptions about where on the plasma panel the map will be and where there will be blank space for targets.

This routine was written before the touch target and print routines were updated. It will need to be updated, along with the other map routines, to use the new routines.

NAME:

mfps

PURPOSE:

Returns the status of the processor. This is an internally coded value which can be used to determine if the processor is currently running in the "interruptable" state.

USAGE:

```
int mfps (), status;
```

```
status = mfps ();
```

PARAMETERS:

None

RETURNS:

A word giving the processor state. If bit 7 (defined as "non_interruptable" in constants.incl) is on, the processor is not interruptable.

CALLS:

None

NOTES:

Complementary routine to mtps.

NAME:

mk_cursor

PURPOSE:

Print or erase a cursor on the panel at the end of the current printing.

USAGE:

mk_cursor (0);
or
mk_cursor (1);

PARAMETERS:

mode Integer 1 to lite, 0 to erase.

RETURNS:

Nothing

CALLS:

get_cursor
get_size_chars
putline

NAME:

mk_page

PURPOSE:

Fills in a page_desc structure with user supplied information describing a page.

USAGE:

```
struct page_desc page;  
int left, bottom, width, height;  
  
mk_page (&page, left, bottom, width, height);
```

PARAMETERS:

p_ptr	Pointer to page_desc	Pointer to structure to be filled in
left	Integer	Coordinate of left edge of page in dots from the left of the screen
bottom	Integer	Coordinate of bottom edge of page in dots from the bottom of the screen
width	Integer	Width of page in dots
height	Integer	Height of page in dots

RETURNS:

Nothing

CALLS:

Nothing

NOTES:

Used to fill in page_desc structure before calling set_page.

All system routines which access page structures assume all coordinates and dimensions are in dots.

NAME:

mtps

PURPOSE:

Sets the processor status to a given value. This is primarily used by IT routines to allow or disallow processor interrupts.

USAGE:

```
int status;
```

```
mtps (status);
```

PARAMETERS:

status

integer

The word to set the processor status to. If bit 7 (defined as `non_interruptable` in `constants.incl`) is on, the processor will not be interruptable. Otherwise, interrupts will be enabled.

RETURNS:

Nothing

CALLS:

None

NOTES:

Complementary routine to `mfps`.

NAME:

open

PURPOSE:

Open does primary maintenance of the device table for the IT. It keeps track of the current owner of the device, and maintains a list of the processes that want to become owner of each device. Open must be called to gain ownership of any device or disk file.

USAGE:

```
int      open();
char     *name;
int      flag, status, file_id;

file_id = open(name, flag, &status);
if (file_id < 0) { error }
```

PARAMETERS:

name A pointer to a character string. This determines the physical device or disk file to open. A set of reserved names have been established for physical devices. These values are defined in constants.incl to be:

<u>device name</u>	<u>value</u>
TOUCH_PANEL	"/dev_tp"
KEYBOARD	"/dev_kb"
PHONE	"/dev_ph"
VIP	"/dev_vip"
DISK0	"/dev_disk0"
DISK1	"/dev_disk1"
DISK2	"/dev_disk2"
DISK3	"/dev_disk3"

(Note that DISK2 and DISK3 are only defined in Level 6 IT systems.)

If name does not match one of these pre-defined names, it is taken to indicate a disk file.

flag An integer flag. If it is non_zero, open will wait for the device to become available. If it is zero, and the device is currently owned, return -1.

status Pointer to an integer that will be set to the status value returned by the device process.

RETURNS:

-1 If the name is invalid, if the device is currently owned and flag is 0 or if the device process indicated a catastrophic failure.

Integer A unique file ID. This value should be used in all subsequent calls to read, write, close, etc.

CALLS:

alloc
close
determine
error
free
pee
vee
write_q

NOTES:

See the discussion of naming conventions on the section I/O System and Device Drivers for a more detailed explanation of the interpretation of file names.

NAME:

parm_xpand

PURPOSE:

To convert a value to a string according to a specified format.

USAGE:

```
char *parm_xpand(), *how, *expanded;  
int *what, count;  
  
expanded = parm_xpand(how, &what, &count);
```

PARAMETERS:

how Pointer to character

This is the format string that determines how the value is to be interpreted. The string pointed to by how should have the format "%x", where x is one of the formats accepted by printf.

what Pointer to pointer to integer

This indicates the parameter to be expanded. It is a pointer to a pointer so that we can update the parameter-pointer in the caller's procedure.

count Pointer to integer

Points to one of the caller's variables that is to be updated to hold the count of the characters used in the current format specification. This allows the caller to skip over the format part in the line expansion. Currently this value is always set to 2 since we don't recognize complex format expressions.

RETURNS:

Pointer to character

Points to the expanded string. The string will be overwritten by subsequent calls to parm_xpand.

CALLS:

str_num

NOTES:

The formats accepted by parm_xpand are the same as the ones described under printf.

This should be fixed to handle long integers. This involves recognizing formats ld, lu, lo, and lx.

NAME:
pathname

PURPOSE:
Compares character strings. YOUR character string will be compared against DIR. The comparison stops on a discrepancy or either of two terminators encountered in YOUR string. The delimiters are NUL (octal 000) and '/'. If the strings were identical, a character pointer is returned pointing at the delimiter in YOUR string. Otherwise a pointer to YOUR is returned.

USAGE:
char *your, *dir, *matched;
char *pathname();

matched = pathname(your,dir);

PARAMETERS:
your pointer to character Pointer to your character string.
dir pointer to character Pointer to the disk file name.

RETURNS:
character pointer Pointer into the "your" string.

CALLS:
nothing

NAME:
 pause

PURPOSE:
 Relinquish the processor temporarily. This is typically done as a courtesy to other processes.

USAGE:
 pause();

PARAMETERS:
 None

RETURNS:
 Nothing

CALLS:
 enq_RQ (kernel)
 block (kernel)

NAME:

pee

PURPOSE:

Pee implements Dijkstra's P primitive. It is used to synchronize two (or more) processes.

USAGE:

```
struct semaphore sem;
```

```
pee (&sem);
```

PARAMETERS:

```
sem_ptr      A pointer to a semaphore
```

RETURNS:

Nothing

CALLS:

```
enq      (kernel)
block    (kernel)
mfps     (kernel)
mtps     (kernel)
```


NAME:

peek

PURPOSE:

Peek is used to determine how much input is currently available from a device. It is intended to eliminate the need for non-blocking read requests. The return value from peek is the number of bytes of data that are available.

USAGE:

```
int peek(), length, device_id, status;
```

```
length = peek(device_id, &status);
```

PARAMETERS:

device_id	An integer that identifies the device to be peeked. This should be the value returned from calling open.
st_ptr	Pointer to an integer that will be set to the device process status.

RETURNS:

-1	If the device_id is invalid, or if the device is not owned by the calling process, or if the device process indicates a catastrophic failure.
data length	Under all other conditions, peek returns the number of bytes of data that is available from the device.
status	The status of the device process is returned indirectly, via st_ptr.

CALLS:

pee	(kernel)
write_q	(kernel)

NAME:

peeper

PURPOSE:

Keeps an eye on the phone line and periodically informs ph_driver of its status.

USAGE: (See NOTES)

This process should never be called as a subroutine.

PARAMETERS:

queue	Pointer to queue_head structure
	The queue to which messages are written.
	Generally, this will be the input queue of the phone driver.

RETURNS:

Never

CALLS:

write_q
pause

NOTES:

Peeper is created as a process by ph_driver. As a process it should not be called by other routines.

This process has a priority of -1. This is lower than all other system processes to ensure that peeper is only invoked if nothing else is ready.

NAME:

ph_driver

PURPOSE:

The phone process allows application programs to use the DLV11 interface to the phone line.

ph_driver works in conjunction with the phone interrupt handlers, phrint and phxint. Input characters from the phone are given by phrint to ph_driver, which saves them until they can be forwarded to an application program. Write requests to the phone are stored up while the transmit interrupt routine does the transmission.

USAGE: (See NOTES)

```
external struct queue_head PP_Q;  
struct req_block rb;  
int special_value;
```

```
write_q (&PP_Q, &rb);
```

or

```
write_q (&PP_Q, special_value); /* special_value must be odd */
```

PARAMETERS:

None in the usual sense. Communication is via the input queue. Items from the queue are either pointers to request blocks, or codes from one of the interrupt routines.

RETURNS:

Never returns. Application program requests are vee'd after they have been handled.

CALLS:

```
read_q (kernel)  
vee (kernel)  
creep (kernel)
```

NOTES:

This routine should never be invoked directly by user programs. They should access this routine thru I/O routines read, write, etc.

This process has a priority of 8.

The phone has been defined to be a line-oriented device. Thus, a read from the phone will return a string containing the minimum of the number of characters requested and the number up to and including the next newline. The returned string is null-terminated.

If a second read or second write comes in before the first has been satisfied, the first read or write will be lost. This is defined to be impossible if the user calls the subroutines read and write. It may happen if the user communicates directly with this process.

Starts up the routine peeper as a process with priority -1 to monitor the state of the communications line.

NAME:

ph_driver

PURPOSE:

The phone process allows application programs to use the MLCP interface to the asynchronous communication ("phone") line.

ph_driver works in conjunction with the phone interrupt handlers, phrint and phxint. Data recieved by phrint is inserted directly into the phone process' buffer (PH_buf) where it is stored until it can be forwarded to an application program. Write requests to the phone are stored up while the MLCP channel program does the transmission. When the write is competed, phxint will notify ph_driver.

USAGE: (See NOTES)

```
external struct queue_head PH_Q;  
struct req_block rb;  
int special_value;
```

```
write_q (&PH_Q, &rb);
```

or

```
write_q (&PH_Q, special_value); /* special_value must be less than 256 */
```

PARAMETERS:

None in the usual sense. Communication is via the input queue. Items from the queue are either pointers to request blocks, or codes from one of the interrupt routines.

RETURNS:

Never returns. Application program requests are vee'd after they have been handled.

CALLS:

```
io  
init_ccb  
restart_io  
disable_io  
read_q  
vee  
creep
```

NOTES:

This routine should never be called directly by user programs. Access should be thru I/O routines read, peek, etc.

This process has a priority of 8.

The phone has been defined to be a line_oriented device. Thus, a read from the phone will return a string containing the minimum of the number of characters requested and the number up to and including the next newline. The returned string is null-terminated.

If a second read or second write comes in before the first has been satisfied, the first read or write will be lost. This is defined to be impossible to the user calls the subroutines read and write. It may happen if the user communicates directly with this process.

NAME:

phrint

PURPOSE:

Phone input interrupt handler

Accepts characters from the phone and forwards them to the phone process.

USAGE:

Called as an interrupt handler only.

PARAMETERS:

None

RETURNS:

Nothing

CALLS:

write_q (kernel)

NAME:

phrint

PURPOSE:

Asynchronous communication line input interrupt routine.

USAGE:

Called as an interrupt handler, only.

PARAMETERS:

None

RETURNS:

Nothing

CALLS:

ch_id
init_ccb
io
write_q

NAME:

phxint

PURPOSE:

Phone output interrupt handler

Writes data to the phone line

USAGE:

Called as an interrupt handler only.

PARAMETERS:

None

RETURNS:

Nothing

CALLS:

write_q (kernel)

NAME:

phxint

PURPOSE:

Asynchronous communications output interrupt routine.

USAGE:

Called as an interrupt routine, only.

PARAMETERS:

None

RETURNS:

Nothing

CALLS:

ch_id
io
write_q

NAME:

pp_write

PURPOSE:

Performs operations on the Level 6 plasma panel by writing specified control information to the panel controller.

USAGE:

```
int code, count;  
char buffer[];
```

```
pp_write (code, buffer, count);
```

PARAMETERS:

code	integer	This is the "task code" word to be given to the panel controller to indicate the type of operation to be performed.
buffer	pointer to character	Pointer to buffer of data to write. This contains parameters for the specific operation.
len	integer	Number of characters to write.

RETURNS:

Nothing

CALLS:

```
io  
iold  
mtps  
mfps  
pause  
rls_pnl  
rsrv_pnl
```


NAME:

pp_read

PURPOSE:

Allows processes to read from the z80 plasma panel controller.
Data read can include such things as the cursor position and the contents of the z80's memory.

USAGE:

```
int code, len;  
char buffer[...];  
  
pp_read (code, buffer, len);
```

PARAMETERS:

code	integer	This is the "task" code word to be given to the panel controller to indicate the type of operation to be performed.
buffer	pointer to character	Pointer to buffer of data to read
len	integer	Number of characters to read.

RETURNS:

Nothing

CALLS:

```
io  
iold  
mtps  
mfps  
pause  
rls_pnl  
rsrv_pnl
```


NAME:

`printf`

PURPOSE:

To print a string on the plasma panel performing formatted parameter replacement within the string.

USAGE:

```
printf("Format string with formats like %d %o, etc, p1, p2, ... );
```

PARAMETERS:

`format` Pointer to character

Points to the format string.

`p1, p2, p3...`

There are an arbitrary number of replacement parameters. One parameter will be used for each format specification encountered in the format string. The type of the specific parameter will be treated as if it matches the type indicated by the format.

Parameter replacement is indicated by a '%' in the format string. Parameter type is specified by the character following the '%'. The following formats are recognized:

d	signed decimal
o	octal (will print a leading 0)
x	hexidecimal (will print a leading X)
l	unsigned decimal
c	character
s	pointer to null-terminated character string

RETURNS:

Nothing

CALLS:

(LSI-11 version)

`parm_xpand`
`put_ascii`

(Level 6 version)

`parm_xpand`
`put_string`

NOTES:

Printf does not accept all the format specifications of the Unix `printf`. In particular, field width and precision specifications are not implemented.

NAME:

`pr_n_clear`

PURPOSE:

Print to the plasma panel while blanking out lines if front of printing.

USAGE:

```
char message[...];
int msg_length;

pr_n_clear (message, msg_length);
           or
pr_n_clear ("Null terminated string", -1);
```

PARAMETERS:

buf	Pointer to character	Points to string of characters to be printed
lth	Integer	Number of chars in buffer. If this number is less than 0, pr_n_clear assumes that the buffer is a standard null-terminated string and prints all characters up to the null.

RETURNS:

Nothing

CALLS:

(LSI-11 version)

- `area_lite`
- `get_cursor`
- `get_size_chars`
- `mk_cursor`
- `printf`

(Level 6 version)

- `mk_cursor`
- `pp_write`

NOTES:

This procedure is not part of the IT system software. As such it may not be revised to be compatible with future versions of the system. The object code for this procedure is not contained in the LSI-11 system library libI.a.

NAME:

put

PURPOSE:

Put is used to perform parallel accesses to the plasma panel. It can selectively write each dot in a group of 16 dots arranged vertically on the panel.

USAGE:

```
int X, Y, scalar, vector[...], count;
```

```
put (X, Y, scalar, count, 0);
```

```
or
```

```
put (X, Y, vector, count, 1);
```

PARAMETERS:

X	An integer that is the X coordinate of the first vector to put. All subsequent vectors will be put at succeeding X coordinates.
Y	An integer that is the Y coordinate of the lower end of the 16 dot vector.
vector	This is either an integer or a pointer to a group of integers. In either case, the bits in the integer(s) which are set will cause the corresponding dot to be lit. The low order bit of the integer corresponds to the lowest dot in the vector.
count	An integer that tells how many vectors to put on the panel. If flag is 0, one vector will be put count times; otherwise, count vectors will be put once each.
flag	An integer that tells how to interpret vector, above. If it is 0, vector is considered an integer and is put out count times. If flag is nonzero, vector is treated as a pointer to an array of integers that is count long.

RETURNS:

Nothing

CALLS:

(LSI-11 version)

Nothing

(Level 6 version)

pp_write

rls_pnl

rsrv_pnl

NOTES:

The bus address (in the LSI version) or the channel address (in the Level 6 version) of the plasma panel interface is "known" by this routine. If the address ever changes, the address defines will have to be changed and put will have to be recompiled.

NAME:

`put_ascii`

PURPOSE:

Print a character on the plasma panel according to the function specified for that character in the current charset. Updates the system cursor according to the character printed.

USAGE:

```
int X, Y;  
char ch;
```

```
put_ascii (ch, &X, &Y);
```

PARAMETERS:

<code>ch</code>	Character	The character to print.
<code>x_ptr =</code>	Pointer to integer	Points to the horizontal coordinate of where to print the character. The value pointed to will be updated depending on the function of <code>ch</code> in the current charset.
<code>y_ptr =</code>	Pointer to integer	Points to the vertical coordinate of where to print the character. The value pointed to will be updated depending on the function of <code>ch</code> in the current charset.

RETURNS:

Nothing
(Returns values indirectly through the parameters.)

CALLS:

<code>area_lite</code>	
<code>ch_to_d</code>	(parameterized define)
<code>d_to_ch</code>	(parameterized define)
<code>d_to_l</code>	(parameterized define)
<code>erase</code>	
<code>get_page_size</code>	(in <code>get_set.c</code>)
<code>l_to_d</code>	(parameterized define)
<code>putchar</code>	
<code>ring_bell</code>	
<code>screen_clear</code>	

NOTES:

The dot pointed to by `x_ptr` and `y_ptr` corresponds to the lower left corner of the printed character.

Formfeed will perform a full screen erase if it determines that the current printing page is "almost" the entire page. If the page is at least a full screen high, and within one character width of each edge then it is determined to be close enough to a full page, and the entire panel is erased.

Should add a page return function which positions the cursor at the upper left corner without clearing the screen. Should consider adding superscript and subscript functions too.

NAME:

putchar

PURPOSE:

Display a character on the plasma panel.

USAGE:

int x, y, ch;

putchar (x, y, ch);

PARAMETERS:

x	Integer	Horizontal coordinate of the lower left corner of where to put the character.
y	Integer	Vertical coordinate of the lower left corner of where to put the character.
ch	Integer	The character to print. The bit masks for each charset are arranged in the same order as the characters they will print.

RETURNS:

Nothing

CALLS:

put

NAME:
putdot

PURPOSE:
Given x and y coordinates of a dot on the plasma panel, this routine either lights or erases that dot. The position is specified in x and y dot addresses, where (0,0) is the lower left corner of the screen.

USAGE:

int x, y, mode;

putdot (x, y, mode);

PARAMETERS:

x	integer	The x coordinate
y	integer	The y coordinate
mode	integer	0 to erase, non-zero to light

RETURNS:
Nothing

CALLS:
(LSI-11 version)
Nothing

(Level 6 version)
pp_write

NAME:

putline

PURPOSE:

Writes or erases a line between two points on the plasma panel.

USAGE:

int xa, ya, xr, yr;

putline (xa, ya, xr, yr, 0); /* erases a line */

putline (xa, ya, xr, yr, 1); /* writes a line */

PARAMETERS:

xa	integer	The x coordinate for the first point
ya	integer	The y coordinate for the first point
xr	integer	The x coordinate for the second point
yr	integer	The y coordinate for the second point
mode	integer	0 to erase line, non-zero to write it

RETURNS:

Nothing

CALLS:

(LSI-11 version)

Nothing

(Level 6 version)

pp_write

NOTE:

The line drawn between two points is unique regardless of the order in which the end points are specified.

NAME:

put_string

PURPOSE:

Prints a string of characters on the remote display head(s).

USAGE:

```
char buffer[...];  
int len;  
  
put_string (buffer, len);
```

PARAMETERS:

buffer	pointer to character	Pointer to a buffer containing the characters to write
len	integer	The number of characters to print

RETURNS:

Nothing

CALLS:

```
ld_cs  
ld_page  
pp_write
```

NOTES:

Format effectors and character generation are handled by the z80 panel controller.

No cursor internal to the Level 6 is updated. In order to find the location of the cursor, programs must call `get_cursor`.

NAME:

read

PURPOSE:

Read transfers data from a device process to a user program.

USAGE:

```
int read(), device_id, buf_len, status, len_in;  
char buffer[...];  
  
len_in = read(device_id, buffer, buf_len, &status);
```

PARAMETERS:

device_id	An integer that identifies the device to read. This should be the value returned by open.
buffer_ptr	Pointer to a user-supplied buffer area large enough to hold the amount of data requested. This is treated as a character pointer, so there is no alignment problem.
length	The number of bytes to read.
st_ptr	Pointer to an integer that will be set to the value of the status of the device process.

RETURNS:

-1	If the specified device is invalid, or if the calling process is not the owner of the device, or if the device process indicated catastrophic failure.
data length	Otherwise, return the number of bytes actually transferred. This may be less than the number requested.
status	See st_ptr under PARAMETERS

CALLS:

write_q	(kernel)
pee	(kernel)

NAME:

read_q

PURPOSE:

Remove an element from a queue. Read_q differs from deq in that read_q will wait for an entry on the queue before it tries to remove one. This eliminates the "deq from empty queue" error.

USAGE:

```
int read_q, q_value;
struct queue_header queue;

q_value = read_q(&queue);
```

PARAMETERS:

q_ptr A pointer to a standard queue header.

RETURNS:

The value deq removed from the queue.

CALLS:

pee	(kernel)
deq	(kernel)

NAME:

refilter

PURPOSE:

Fill in a map outline with the appropriate shadings.

USAGE:

```
int val[...], old_val[...];
```

```
refilter (val, old_val);
```

PARAMETERS:

val	Array of integers
	Number of the new shade for each county
old_val	Array of integers
	Number of the shade currently in each county

RETURNS:

Nothing

CALLS:

erase

put

NOTES:

This is one the the relatively obscure parts of the IT map stuff. This will all most likely be rewritten before it is released (to make use of the new print and touch target stuff, and to make it generally more readable). As a result, this documentation is left somewhat sketchy, assuming it will need to be redone anyway.

NAME:

restart_io

PURPOSE:

Restore the non-zero elements of the LCT for an MLCP channel, and enable input or output on that channel.

USAGE:

int chan, it_id;

if (restart_io (chan, it_id) != 0) { error }

PARAMETERS:

chan	Integer	Address of the MLCP channel to be restarted.
it_id	Integer	IT id of the line (ie, device) that includes this channel.

RETURNS:

0	If everything worked alright.
-1	If any errors were encountered talking to the MLCP.

CALLS:

io
enable_io

NAME:

ring_bell

PURPOSE:

Ring the bell on the touch panel.

USAGE:

ring_bell ();

PARAMETERS:

None

RETURNS:

Nothing

CALLS:

(LSI-11 version)

Nothing

(Level 6 version)

pp_write

NAME:

rls_pnl

PURPOSE:

This routine is used by a process to release the remote display heads to other processes.

USAGE:

rls_pnl ();

PARAMETERS:

None

RETURNS:

0	if the remote display heads were freed
-1	if the display heads were not owned by this process

CALLS:

vee

NOTES:

Complementary routine to rsrv_pnl.

Remote display head reserves (calls to rsrv_pnl) from the same process are stacked. If two calls are made to rsrv_pnl by a process, it will take two calls to rls_pnl to release the display heads. This is so that a subroutine can reserve and release the display heads without worrying whether or not the calling routine has already reserved them.

NAME:

rsrv_pnl

PURPOSE:

This routine is called by to grant a process exclusive access to the remote display heads. It is used to ensure that multiple processes do not interfere with each other when doing i/o to the heads.

USAGE:

rsrv_pnl ();

PARAMETERS:

None

RETURNS:

Nothing

CALLS:

pee

NOTES:

Complementary routine to rls_pnl.

Remote display head reserves (calls to rsrv_pnl) from the same process are stacked. If two calls are made to rsrv_pnl by a process, it will take two calls to rls_pnl to release the display heads. This is so that a subroutine can reserve and release the display heads without worrying whether or not the calling routine has already reserved them.

NAME:

s_read

PURPOSE:

Read one sector from physical disk.

USAGE:

```
int    drv, sec;
char   *ubuf;
int    status;
```

```
status = s_read(drv, sec, ubuf);
```

PARAMETERS:

drv	integer	• Drive to be read from.
sec	integer	Logical sector number.
ubuf	pointer to character	Indicates the user's 128 byte buffer to be filled with data read. All 128 bytes are used.

RETURNS:

-1	Error.
0	No error.

CALLS:

```
(LSI-11 version)
swab
```

```
(Level 6 version)
io
iold
swab
```


NAME:

s_write

PURPOSE:

Write one sector to physical disk.

USAGE:

```
int    drv,sec,status;  
char   *ubuf;
```

```
status = s_write(drv, sec, ubuf);
```

PARAMETERS:

drv	integer	Drive to be written on.
sec	integer	Logical sector number.
ubuf	pointer to character	Indicates the user's 128 byte buffer where data for the disk resides.

RETURNS:

0	No error.
-1	Error.

CALLS:

(LSI-11 version)

swab

(Level 6 version)

```
io  
iold  
swab
```


NAME: save_free

PURPOSE: Save the "in memory" version of the freemap on the disk.

USAGE:

```
int      drv, status;

status = save_free(drv);
```

PARAMETERS:

drv	integer	Number of disk drive.
-----	---------	-----------------------

RETURNS:

0	No error.
-1	Error.

CALLS:

bufio

NAME:

`screen_clear`

PURPOSE:

Clear the plasma panel

USAGE:

`screen_clear ();`

PARAMETERS:

None

RETURNS:

Nothing

CALLS:

(LSI-11 version)

Nothing

(Level 6 version)

`pp_write`

NAME:
scrunch

PURPOSE:
To remove one or more adjacent elements of an array and scrunch the remaining pieces back together.

USAGE:
int array[...];

scrunch (array+first_deleted,
 size-num_deleted-first_deleted,
 num_deleted);

/* from a array with 'size' entries, this should produce an array with
size-num_deleted entries. A group of 'num_deleted' entries starting
with 'first_deleted' will be squeezed out */

PARAMETERS:

ptr	Pointer to integer	The starting address of the hole to be filled in
size	Integer	Number of elements in the last piece of array
num	Integer	The number of elements to eliminate in the array (or the number of places over to move)

RETURNS:
Nothing

CALLS:
Nothing

NAME:
 set_charset

PURPOSE:
 Sets the current charset for printing.

USAGE:
 struct cs_desc cs;

 set_charset (&cs);

PARAMETERS:
 cs_p Pointer to cs_desc structure
 Points to new charset to be used.

RETURNS:
 Nothing

CALLS:
 Nothing

NOTES:
 Complimentary function to get_charset.

NAME:

`set_cursor`

PURPOSE:

Positions the cursor on the current page. The position is specified in character coordinates, using the current character set.

USAGE:

`int x, y;``set_cursor (x,y);`

PARAMETERS:

<code>x</code>	Integer	Horizontal position of the cursor (in character spaces)
<code>y</code>	Integer	Vertical position of the cursor (in character lines)

RETURNS:

`Nothing`

CALLS:

`Nothing`

NOTES:

Complimentary function to `get_cursor`.

NAME:

set_env

PURPOSE:

Sets the global descriptors for the charset, cursor position, and page.

USAGE:

struct env_desc env;

set_env(&env);

PARAMETERS:

env_ptr	Pointer to env_desc structure
	Points to the structure which has all
	the values specifying the new environment.

RETURNS:

Nothing

CALLS:

set_charset

set_cursor

set_page

NOTES:

Complimentary function to get_env.

NAME:

set_mode

PURPOSE:

Set_mode is a "black hole" entry into the I/O system. It is used to perform device-dependant operations; the definition of the operations and the parameters they require are left to each device.

USAGE:

```
int set_mode(), device_id, length, status, ret_val;
char mode[...];

ret_val = set_mode (device_id, mode, length, &status);
```

PARAMETERS:

device_id	An integer that specifies the device to operate upon. This should be the value returned from the call to open that acquired the device.
mode_ptr	A pointer to a byte-aligned buffer. The contents of the buffer and its size are device dependant.
length	An integer; the number of bytes pointed to by mode_ptr.
st_ptr	Pointer to an integer that will be set to the value of the device process status.

RETURNS:

-1	If the device_id is invalid, the caller is not the owner of the device, or the device process indicated a catastrophic failure.
Integer	Otherwise, return a device-dependant integer.

CALLS:

write_q	(kernel)
pee	(kernel)

NAME:

set_page

PURPOSE:

Sets the global page descriptor. This determines the area of the plasma panel ("page") which will be used for printing.

USAGE:

```
struct page_desc page;  
int left, bottom, width, height;  
  
mk_page (&page, left, bottom, width, height);  
.  
.  
.  
set_page (&page);
```

PARAMETERS:

page	Pointer to page_desc structure
	Points to a structure containing the values describing the new page.

RETURNS:

Nothing

CALLS:

Nothing

NOTES:

Complimentary function to get_pg.

NAME:

set_pnl

PURPOSE:

This routine is used to specify which of the three plasma panels should be effected by subsequent panel routines. Any combination of the three can be specified.

USAGE:

```
int ppu;
```

```
set_pnl (ppu);
```

PARAMETERS:

```
ppu      integer
```

A coded indicator specifying which panel(s) to use. A value of 1 means use panel 0, 2 means use panel 1, and 4 means use panel 2. These values are additive, so a parameter value of 5, for instance, means that panels 0 and 2 should be used.

RTURNS:

Nothing

CALLS:

None

NAME:
sh_map

PURPOSE:
Add the legend and shade the counties on an outlined map.

USAGE:
int num_shades, val[...], old_shades[...], num_vals, master;

sh_map (num_shades, val, old_shades, num_vals, master);

PARAMETERS:

num_shades	Integer	Number of shades to use on the map
val	Array of integers	Data values for each county
old_shades	Array of integers	Number of the shade used on each county the last time it was shaded. If the map is reshaded, this array will be updated to hold the current shadings upon return.
num_vals	Integer	Number of parcels to shade
master	Integer	Flag that indicates whether need to do the legend or shading or both. Values of legend_no_map, map_no_legend and map_w_legend are recognized. These values are defined in map_defines.incl to be 1, 0, and 2, respectively.

RETURNS:
Nothing
(Returns values indirectly through parameters.)

CALLS:

legend	(map routine)
printf	
refilter	(map routine)

NOTES:
This routine may need to be cleaned up when all the map stuff is updated to use the new touch target and print routines.

NAME:

startup

PURPOSE:

This routine performs the one-time initializations necessary to start up an application system. It initializes the stack and starts up an initial set of processes.

USAGE:

This is not called by any routine. It can be re-entered in the LSI11 version (thus restarting the system) by branching to location 0. Because global variables cannot be re-initialized in the Level 6 version, systems must be re-started in the Level 6 by reloading them from disk.

PARAMETERS:

hicore pointer to int (Level 6 version, only)
Address of highest available word of memory.
This parameter is created by a short piece of code in low, called "entry", which is entered at system start-up.

The processes to be started are contained in PROCTAB, which has the following format:

```
extern int (*procedure_name)();

int PROCTAB[] {
    procedure_name, size_of_stack, parameter, priority, 0,
    .
    .
    .
    0, 0, 0, 0, 0
};
```

where procedure_name is the name of the process procedure, size_of_stack is the size (in words) of the stack to allocate to the process, parameter is a one-word value that is passed to procedure_name, priority is the priority of the process (the bigger the number, the higher the priority), and the trailing zero will be replaced by the ID of the process. The last entry in PROCTAB must consist of all zero's.

RETURNS:

Never returns. Exits by a call to first_block.

CALLS:

(LSI-11 version)

alloc
creep
first_block
io_init
mk_page
printf
screen_clear
set_charset
set_page

(Level 6 version)

alloc
creep
first_block
init_pnl
io_init
io
ld_mlep
mk_page
ld_mlep
printf
screen_clear
set_charset
set_page
set_pnl

NAME:

`str_num`

PURPOSE:

Converts a single precision unsigned integer into its ASCII representation.

USAGE:

```
int num, base;
char *place;
```

```
str_num(num, base, &place);
```

PARAMETERS:

num	Integer	The number to convert.
base	Integer	The base to use for conversion. Currently, this must be between 2 and 16.
place	Pointer to pointer to character	Pointer to character pointer that tells where the output is to go. Is updated so that on return place points one character after the end of the expanded string.

CALLS:

```
ldiv
lrem
str_num          (self)
```

NOTES:

Complementary function to cvb.

The expanded string is not null terminated. It probably should be.

The maximum base size can be changed by updating the defined value for Highest_Base and the characters in the possible_digits defined string.

Should be fixed to do double precision integers, too.

NAME:

swab

PURPOSE:

Swap the bytes of the words in an integer array. High bytes and low byte are exchanged in each integer.

USAGE:

```
int array_pt[...], length;
```

```
swab (array_pt, length);
```

PARAMETERS:

array_pt	pointer to an array of integers
length	integer length of the array, in words

RETURNS:

Nothing

CALLS:

Nothing

NOTES:

The usage of this routine is different from that of the Level 6 routine with the same name.

NAME:

swab

PURPOSE:

Swap bytes in an array of characters. The first and second bytes are exchanged, the third and fourth are exchanged, etc.

USAGE:

```
char buffer[...];  
int length;  
  
swab (buffer, length);
```

PARAMETERS:

buffer	Pointer to array of characters to swap.
length	The integer number of pairs of characters to swap.

RETURNS:

Nothing

CALLS:

None

NOTES:

The usage of this routine is different from that of the LSI-11 routine by the same name.

NAME:

tp_driver

PURPOSE:

The touch panel driver allows application programs to read touches from the touch panel.

tp_driver works in conjunction with the touch panel interrupt routine, tp_int. It accepts data from the interrupt handler and buffers it until requested by an application program.

USAGE: (See NOTES)

```
external struct queue_head TP_Q;  
struct req_block rb;  
int special_code;
```

```
write_q (&TP_Q, &rb);
```

or

```
write_q (&TP_Q, special_value); /* special_value must be odd in the  
LSI-11 version and less than 256 in  
the Level 6 version */
```

PARAMETERS:

None in the usual sense. Communication is via the input queue. Items on the input queue are either addresses from the interrupt routine, or pointers to request blocks.

RETURNS:

Never returns. When application program requests have been processed, they are vee'd.

CALLS:

(LSI-11 version)

```
read_q (kernel)  
vee (kernel)
```

(Level 6 version)

```
get_pnl  
pp_write  
rls_pnl  
rsrv_pnl  
read_q (kernel)  
set_pnl  
vee (kernel)
```

NOTES:

This routine should never be accessed directly by user programs. Access should be thru the routines read, peek, etc.

This process has a priority of 8.

A read from the touch panel will return an even-length string containing the minimum of the number of bytes requested and the number of bytes currently in the touch panel buffer. If the buffer is empty, the next set of coordinates from the touch panel is returned. After the read, the x-coordinate of the touch is in byte 0 of the user's buffer

and the y-coordinate is in byte 1.

If a second read comes in before the first has been processed, the first will be lost. This is defined to be impossible if the user calls the subroutine read. It may happen if the user communicates directly with this process.

NAME:

tp_interrupt

PURPOSE:

This is the interrupt handler for the touch panel. It forwards 'touches' to the touch panel process, which eventually gives them to the application programs.

USAGE:

Entered as an interrupt processor only.

PARAMETERS:

None

RETURNS:

Nothing

CALLS:

write_q

NAME:

Trap

PURPOSE:

Intercepts traps in the LSI-11 IT. Trap prints a message that a trap occurred, and then calls error.

USAGE:

Entered when a trap occurs.

PARAMETERS:

None

RETURNS:

Never returns

CALLS:

error

NAME:

TRPHND

PURPOSE:

Intercepts unavailable resource traps in the Level 6 IT.

CALLS:

error

NOTES:

This routine was included to circumvent an apparent hardware problem somewhere in the interface to the z80 remote display head controllers. For some reason, the software occasionally gets unavailable resource traps when doing an io or iold instruction to the z80. This routine checks each trap to see if it was caused by an io or iold to one of the panels. If it was, it restarts the processor at the offending instruction. The instruction will eventually be accepted. If the trap is caused by any other problem, TRPHND calls error.

This routine assumes that the offending instructions were issued by the standard io or iold routines. These routines copy the channel address into register R6.

NAME:

tt_activate

PURPOSE:

Activate a target variable. This includes displaying the target (where applicable) and adding a pointer to the target variable to the list of active targets.

USAGE:

```
int tt_activate(), slot;  
struct target t;  
  
slot = tt_activate(&t);
```

PARAMETERS:

t_p	Pointer to a target structure
	Used to access the target variable containing the description on the target to be activated.

RETURNS:

slot	The index in tt_current of the element which now points to the target variable. This index can be passed to other target-manipulating procedures to work on this active target.
-1	Error return, indicating that the array of active targets is full.

CALLS:

```
tt_label  
tt_outline
```


NAME:
tt_arranger

PURPOSE:
Allows application programs to create a collection of related targets in one area of the screen. The calling program must specify some specific attributes for each target, and some attributes for the group as a whole. From these specifications, the procedure will call tt_create to fill in user-supplied target variables with the necessary values to make targets in the appropriate positions.

USAGE:

```
struct target list[...];
int num, values[...], flags[...], t_width, t_height, mode;
char *labels[...];
struct cs_desc *cs[...];
struct page_desc area;

tt_arranger (list, num, values, labels, cs, flags, &area,
            t_width, t_height, mode);
```

PARAMETERS:

list	Array of target structures	These get filled in by tt_create to describe the new targets.
num	Integer	Number of targets to position
values	Array of integers	Values for created targets
labels	Array of pointers to characters	Labels for created targets
cs	Array of pointers to cs_desc structures	Identifiers for charsets to use for created targets
flags	Array of integers	Flag values for created targets These are the same as for tt_create.
area_p	Pointer to page_desc structure	The pointed-to page_desc variable will be taken to describe the area of the screen which the group of targets will fill.
t_width	Integer	Width of each target in the group (in dots)
t_height	Integer	Height of each target in the group (in dots)
mode	Integer	Indicates positioning of the group, as a whole, within the specified area. If vert_cent bit is on, the group will be centered vertically in the area. Otherwise, it will be placed at the top of the area. If the horiz_cent bit is on, group will be centered horizontally. Otherwise, it will be left-justified. The values for vert_cent and horiz_cent are defined in tgt_const.incl to be 2 and 1, respectively.

RETURNS:

Nothing

CALLS:

tt_create

NOTES:

There is no check to verify that the specified number of targets of the specified size will fit in the specified area. If the group is too large for the area, something unknown and bizarre will happen.

The last row of targets may not be correctly centered, as all centering calculations use truncating divisions.

NAME:

tt_cleanup

PURPOSE:

To unconditionally deactivate all active targets.

USAGE:

tt_cleanup();

PARAMETERS:

None

RETURNS:

Nothing

CALLS:

tt_delete

NAME:

tt_create

PURPOSE:

Fill in the fields of a target variable to describe a specific touch target.

USAGE:

```
struct target t;
int x, y, width, height, value, flag;
char *label;
struct cs_desc cs;
```

```
tt_create (&t, x, y, width, height, value, label, &cs, flag);
```

PARAMETERS:

t_p	Pointer to target structure	This structure will be filled in by the procedure.
x	Integer	Horizontal coordinate of the lower left corner of the touch target (in dots)
y	Integer	Vertical coordinate of the lower left corner of the touch target (in dots)
width	Integer	Width of the touch target (in dots)
height	Integer	Height of the touch target (in dots)
value	Integer	Value of the touch target. This is the target value which is returned by tt_selections. It can be used by applications programs for internal identification of the target.
label	Pointer to character	Points to the text used to label this target. A value of 0 indicates no label.
cs_p	Pointer to cs_desc structure	Indicates the character set to use when labelling the target. This value is meaningless if label is 0.
flag	Integer	Flags which determine the attributes of this target. The specific bits used for each function is specified by the XX_bit defines in tgt_const.incl. The types of functions specified are:

<u>bit</u>	<u>meaning</u>
0: label_bit	off : normal on : do not label this target used by : tt_label
1: outline_bit	off : normal on : do not outline this target used by : tt_outline
2: flash_bit	off : normal on : do not flash this target used by : tt_flash, tt_read
3: mark_bit	off : normal on : do not mark this target when it is selected used by : tt_selections
4: frc_rtn_bit	off : normal

	on	:	reading this target will force tt_selections to return
	used by	:	tt_selections
5: dup_hits_bit	off	:	duplicate hits of this target will cancel each other
	on	:	successive hits of this target will have no effect on each other
	used by	:	tt_selections
6: other_bit	off	:	normal
	on	:	cause tt_selections to increment count of max number of targets to select
	used by	:	tt_selections
7: in_use	off	:	target variable not active
	on	:	target variable active
	used by	:	set by tt_activate, reset by tt_delete
8: lite_bit	off	:	normal
	on	:	do not lite or erase this target
	used by	:	tt_lite
13: tt_pnl0	off	:	target not displayed on remote display head 0
	on	:	target is displayed on remote display head 0
	used by	:	tt_label tt_lite tt_mark tt_outline tt_read
			(See NOTES)
14: tt_pnl1			Same as tt_pnl0, but causes target to display on remote display head 1
15: tt_pnl2			Same as tt_pnl0, but causes target to display on remote display head 2

RETURNS:

Nothing

(Returns values indirectly through the parameter t_p.)

CALLS:

Nothing

NOTES:

The three flags bits tt_pnl0, tt_pnl1, and tt_pnl2 are used only in the Level 6 version of the IT system. In that version, any combination of the three bits may be lit, so the target may be displayed on any of the three remote display heads. If none of the three bits is lit, then it is assumed that the target should be displayed on head zero, only.

NAME:

tt_deactivate

PURPOSE:

Opposite function of tt_activate. Erases a target and removes it from the list of active targets.

USAGE:

```
int slot;  
  
if (tt_deactivate (slot) < 0) { error }
```

PARAMETERS:

slot	Integer	Index into tt_current of pointer to target variable to be deleted. This corresponds to the return value from tt_create.
------	---------	---

RETURNS:

-1	If slot is not a valid index into tt_current or if tt_current [slot] is 0
0	Otherwise

CALLS:

```
tt_delete  
tt_lite
```


NAME:
tt_delete

PURPOSE:
Make a target variable inactive by removing the pointer to it from the array of active targets (tt_current). Does not erase the displayed target.

USAGE:
int tt_delete(), slot;

if (tt_delete(slot) < 0) { error }

PARAMETERS:
slot Integer Index into tt_current of pointer to target variable to be deleted. This corresponds to the return value from tt_create.

RETURNS:
-1 If slot is not a valid index into tt_current or if tt_current [slot] is 0
0 Otherwise

CALLS:
Nothing

NAME:

tt_flash

PURPOSE:

Flash an active target. Will lite and then erase the entire target area, then re-display the target.

USAGE:

int slot;

if (tt_flash (slot) < 0) { error }

PARAMETERS:

slot Integer Index into tt_current of structure
 describing target to be flashed.

RETURNS:

-1 If slot is out of range, or tt_current [slot] is empty.
0 Otherwise.

CALLS:

tt_label
tt_lite
tt_outline

NAME:

tt_label

PURPOSE:

Display the text label for a target variable.

USAGE:

```
int tt_label (), slot;
```

```
if (tt_label (slot) < 0) { error }
```

PARAMETERS:

slot	Integer	Index into tt_current of the pointer to the target variable to be labeled. This corresponds to the value returned by tt_activate.
------	---------	---

RETURNS:

-1	If either slot is out of range or tt_current [slot] is 0.
0	Otherwise.

CALLS:

(LSI-11 version)

```
get_env  
get_page_size  
get_size_chars  
mk_page  
set_charset  
set_cursor  
set_env  
set_page  
tok_print
```

(Level 6 version)

```
get_env  
get_page_size  
get_size_chars  
mk_page  
rls_pnl  
rsrv_pnl  
set_charset  
set_cursor  
set_env  
set_page  
tok_print
```


NAME:

tt_lite

PURPOSE:

Turns on (or off) all the dots on the panel in the area specified for a target variable.

USAGE:

```
int slot, mode, tt_lite ();

if (tt_lite (slot, mode) < 0) { error }
```

PARAMETERS:

slot	Integer	Index into tt_current of the pointer to the target variable to be lit. This corresponds to the value returned by tt_activate for this target.
mode	Integer	0 if the area is to be erased; non-zero if all the dots are to be turned on.

RETURNS:

-1	Error condition indicating that either slot is out of range or that tt_current [slot] is 0.
0	Normal return

CALLS:

(LSI-11 version)

area_lite

(Level 6 version)

area_lite

get_pnl

rls_pnl

rsrv_pnl

set_pnl

NOTES:

This routine used to be called tt_flash.

NAME:
tt_mark

PURPOSE:
To visually mark a selected touch target, or to erase such a mark.

USAGE:
int tt_mark (), slot, mode;

if (tt_mark (slot, mode) < 0) { error }

PARAMETERS:

slot	Integer	Index into tt_current of a pointer to the target variable to be marked.
mode	Integer	non-zero indicates that the mark should be written ; 0 indicates erase.

RETURNS:

-1	Indicates that either slot is out of range, or that tt_current [slot] is 0
0	Otherwise

CALLS:

(LSI-11 version)

area_lite

(Level 6 version)

area_lite
get_pnl
rls_pnl
rsrv_pnl
set_pnl

NAME:

tt_move

PURPOSE:

Change the t_x and t_y fields of a target variable without specifying values for the other fields.

USAGE:

```
int new_x, new_y;  
struct target t;  
  
tt_move (&t, new_x, new_y);
```

PARAMETERS:

t_p	Pointer to target structure	Points to target variable to be updated
new_x	Integer	If non-negative, specifies the new left edge of the target, in dots. If negative, the t_x field is not changed.
new_y	Integer	If non-negative, specifies the new bottom edge of the target, in dots. If negative, the t_y field is not changed.

RETURNS:

Nothing

CALLS:

Nothing

NOTES:

There is no check made for invalid x and y values.

NAME:

tt_outline

PURPOSE:

Display the outline for a target variable.

USAGE:

```
int tt_outline (), slot, mode;
```

```
if (tt_outline (slot, mode) < 0) { error }
```

PARAMETERS:

slot	Integer	Index into tt_current of the pointer to the target variable to outline. This is the value returned by tt_activate for this target.
mode	Integer	1 to draw the outline; 0 to erase it.

RETURNS:

-1	Indicates that either slot is out of range, or that tt_current [slot] is 0.
0	Otherwise.

CALLS:

(LSI-11 version)
putline

(Level 6 version)
get_pnl
putline
rls_pnl
rsrv_pnl
set_pnl

NAME:

tt_read

PURPOSE:

Allows application programs to read user touches in terms of the set of currently active targets.

USAGE:

```
int tt_read (), fid, slot;
```

```
slot = tt_read (fid);
```

PARAMETERS:

touch	Integer	The file id of the touch panel; returned by open for the touch panel.
-------	---------	---

RETURNS:

slot	Integer	Index into tt_current of description of touched target.
------	---------	---

CALLS:

flush	(I/O system)
read	(I/O system)
tt_flash	
flush	(I/O system)

NAME:

tt_relabel

PURPOSE:

Change the label and value fields of a target variable without specifying the other fields of the variable.

USAGE:

```
int slot, value;
struct target t;
char *str;

tt_relabel(&t, str, value, ptr_mode);
      or
tt_relabel(slot, str, value, slot_mode);
```

PARAMETERS:

t_p	Integer OR pointer to a target structure	Used to access the target variable to be relabeled.
str	Pointer to character	The new label for the target
value	Integer	New value for the target
mode	Integer	A value of slot_mode indicates that t_p is an index into tt_current. Otherwise t_p is taken to be a pointer to a target variable. Slot_mode and ptr_mode are defined in tgt_const.incl to be 0 and 1, respectively.

RETURNS:

Nothing

CALLS:

Nothing

NOTES:

No check is made to verify that a slot-type parameter t is within range.

NAME:

tt_selections

PURPOSE:

Allows application programs to read a number of user touches at once. This procedure returns as complete a list of user touches as possible. It will delete duplicate touches from that list and limit the maximum number of touches in the list, where applicable. Targets whose values are in the list will be marked.

USAGE:

```
int tt_selections(), numt;  
int touch, max_num_tchs, num, ovrfl, values[...], slots[...];  
  
numt = tt_selections(touch, max_num_tchs, num, ovrfl, values, slots);
```

PARAMETERS:

touch	Integer	File id of the touch panel, returned by open.
max_num_tchs	Integer	Maximum allowable number of user touches. Exceeding this number will either cause a target to be deleted from the selected list or the procedure to return, depending on the value of ovrfl.
num	Integer	Number of touches already recorded in the list.
ovrfl	Integer	Indicates what action to take when the user makes more than the maximum allowable number of touches. If ovrfl equals the define ovrfl_rtn, the procedure will return. Otherwise it will delete a target from the list. Ovrfl_rtn is defined in tgt_const.incl to be 0.
values	Array of integers	The list that holds the values for the selected targets
slots	Array of integers	Holds the list of indices into tt_current of the targets that have been selected. These are used for unmarking targets which are removed from the selected list.

RETURNS:

num The number of chosen targets. The values for these targets are stored in the values array, and their corresponding indices into tt_current are stored in slot.

(Returns values indirectly thru parameters.)

CALLS:

scrunch
tt_mark
tt_read

NAME:

vee

PURPOSE:

Vee implements the generalized Dijkstra V primitive.

USAGE:

struct semaphore sem;

vee (&sem);

PARAMETERS:

sem_p A pointer to the semaphore to V

RETURNS:

Nothing

CALLS:

deq	(kernel)
enq_RQ	(kernel)
mfps	(kernel)
mtps	(kernel)

NAME:

verify

PURPOSE:

This is the PL/1 verify function. It takes two strings and returns the index in the first string of the first character which is not in the second string.

USAGE:

```
char *s1, *s2;  
int verify(), ndx;  
  
ndx = verify(s1, s2);
```

PARAMETERS:

s1	pointer to character	Pointer to the source string
s2	pointer to character	Pointer to verification string

RETURNS:

int - Index of first character in the first string which is not in the second string, or -1 if all characters in first string are in second string.

CALLS:

None

NAME:

vip_proc

PURPOSE:

The vip process allows application programs to use the interface to the DN355. It works with the interrupt routines vipxint and viprint (in the LSI-11 version) or the interrupt routine vipint (in the Level 6 version) to simulate a VIP7705 terminal.

Input transmissions are parsed and checked for validity by the routine viprint (or vipint). Input characters are buffered by that routine and are passed on to application programs by vip_proc. Output from application programs is put into the appropriate format by vip_proc and stored in an output buffer. Viprint (or vipint) decides when the output transmission can be initiated. In the LSI-11 version, vipxint does the actual transmission. In the Level 6 version, vipint initiates a DMA output operation.

USAGE: (See NOTES)

```
extern struct queue_head VIP_Q;
struct req_block rb;
int special_value;
```

```
write_q (&VIP_Q, &rb);
```

```
or
```

```
write_q (&VIP_Q, special_value);
```

```
/* special_value must be odd
   in the LSI-11 version and
   less than 256 in the Level
   6 version */
```

PARAMETERS:

None in the usual sense. Communication is via the input queue. Items from the queue are either pointers to request blocks, or codes from one of the interrupt routines.

RETURNS:

Never returns. Application programs requests are vee'd after they have been handled.

CALLS:

(LSI-11 version)

```
mfps      (kernel)
read_q    (kernel)
mtps      (kernel)
vee       (kernel)
```

(Level 6 version)

```
init_ccb
read_q    (kernel)
restart_io
vee       (kernel)
```

NOTES:

If a second read or write comes in before the first has been processed, the first will be lost. This is defined to be impossible if the user calls the subroutines read and write. It may happen if the user

communicates directly with this process.

It is possible for the line to get out of synch. A real vip terminal has a time-out capability which will cause it to send a quiescent frame whenever there has been no activity on the line for about four seconds. This allows recovery if there was some temporary line failure that caused a complete message to be lost. There is no time-out facility in the IT system, so this capability is not present. In the Level 6 version, closing and re-opening the vip line should re-start things. No such provision is provided in the LSI-11 version. It should probably be added.

The "special_value" parameter in USAGE is a code generated by one of the interrupt routines to give information to vip_proc. The codes currently recognized are:

- 3 - An input transmission has been received. If there is a pending read, it can now be satisfied.
- 5 - An ack was recieved. Data has been successfully transmitted, so the currently pending write request can be vee'd.
- 11 - The vip line has just become ready.
- 13 - The vip line has just become not ready.

This process has a priority of 8.

The vip interface is defined as a transmission-oriented device. Thus, a read from the vip will return a null-terminated string which contains the number of characters requested, or all of the characters up to the end of one transmission, whichever is smaller.

NAME:

vipint

PURPOSE:

This is the interrupt service routine for the Level 6 version of the IT vip terminal simulator.

USAGE:

This routine is invoked as an interrupt processor, only.

PARAMETERS:

None

RETURNS:

Nothing

CALLS:

error
init_ccb
io
restart_io
write_q

NAME:

viprint

PURPOSE:

This is the input interrupt processor for the DUV11 vip-simulating interface. It parses input transmissions and keeps a general eye on the state of the interface.

USAGE:

This routine is entered as an interrupt processor, only.

PARAMETERS:

None

RETURNS:

Nothing

CALLS:

error (kernel)

write_q (kernel)

NOTES:

See the write-up of vip_proc for a more complete description of the vip interface.

NAME:

vipxint

PURPOSE:

Controls transmission to the DUV11.

USAGE:

This routine is entered as an interrupt processor, only.

PARAMETERS:

None

RETURNS:

Nothing

CALLS:

None

NOTES:

See the write-up of vip_proc for a more complete description of the vip interface.

NAME:
write

PURPOSE:
Write is the user-level write interface to the I/O system.

USAGE:

```
int write(), len_written, device_id, length, status;  
int buffer[...];  
  
len_written = write(device_id, buffer, length, &status);
```

PARAMETERS:

device_id	An integer that specifies the device to write. This should be the value returned from the call to open that acquired the device.
buf_ptr	A pointer to the data to be written.
length	An integer that is the number of bytes of data to write.
st_ptr	A pointer to an integer that will be set to the value of the status of the device process.

RETURNS:

-1	If the device_id is not valid, or if the device is not owned by the calling process, or if the device process failed catastrophically.
Number of bytes of data written	Under all other circumstances.

CALLS:

write_q	(kernel)
pee	(kernel)

NAME:

write_q

PURPOSE:

Write_q adds an element to a standard queue. It is a counterpart to read_q, and should be used in conjunction with it.

USAGE:

```
struct queue_header q;  
int value;  
  
write_q (&q, value);
```

PARAMETERS:

q_ptr	A pointer to a standard queue header.
value	The item to add to the queue.

RETURNS:

Nothing

CALLS:

enq	(kernel)
vee	(kernel)

NAME:
 xopen

PURPOSE:
 Initialize the file information block given directory information.

USAGE:
 struct file_info fib;
 int index_b, o;
 struct dir_entry slot;

 xopen (&fib, index_b, o, &slot);

PARAMETERS:
 fib pointer to file_info block to be filled in.
 index_b integer index block of directory
 o integer offset in directory
 slot pointer to a dir_entry structure

RETURNS:
 0 always

CALLS:
 nothing

NAME:
 zero_blk

PURPOSE:
 Writes zeros to a disk block.

USAGE:
 struct buffer *ev;
 int blk;
 int status;

 status = zero_blk (ev, blk);

PARAMETERS:
 ev pointer to buffer structure
 Buffer through which the disk may be accessed.
 blk integer The block to be zeroed.

RETURNS:
 0 No error.
 -1 Error.

CALLS:
 bufio
 s_write

NAME:

Z80_LD

PURPOSE:

Called by init_pnl to get a pointer to the z80 remote display head microcode.

USAGE:

```
struct {
    int length;
    int start;
    char microcode [2];    /* this is really "length" characters
                           long */
} *ptr, *Z80_LD ();

ptr = Z80_LD ();
```

PARAMETERS:

None

RETURNS:

Pointer to a structure which contains:

length	integer	Length of the z80 microcode, in bytes.
start	integer	Address at which the z80 should be started
microcode character		"length" characters, containing the Z80 microcode

CALLS:

None

NAME:

z80_int

PURPOSE:

This is the interrupt routine for the Zilog Z80 plasma panel, keyboard, and touch panel controller.

USAGE:

Called as an interrupt handler, only.

PARAMETERS:

channel	integer	The channel address of the device that caused the interrupt
---------	---------	---

RETURNS:

Nothing - Entered as interrupt routine, only

CALLS:

iold
io
vee
write_q

NAME:
 zero_sim

PURPOSE:
 This routine will be called by fio when an unallocated disk block is read (hole). It does the same error checking and read truncation as bufio.

USAGE:
 int d1,d2,o,l,fcn;
 char *userbuf;

 zero_sim (d1, d2, o, userbuf, l, fcn)

PARAMETERS:
 d1 integer A dummy parameter.
 d2 integer Another dummy.
 o integer Offset in block.
 userbuf pointer to character
 User's buffer to be filled with zeros.
 l integer Length of read.
 fcn integer must be 0 (READ)

RETURNS:
 0 No error.
 -1 Error.

CALLS:
 nothing

APPENDIX C

Description of IT Data Structures

NAME:

buffer

PURPOSE:

The buffer structure is used to define the environment seen by disk block IO routines. It holds the physical number of the disk drive, the current block number being used, a sector offset in that block (used for buffering), a flag to indicate whether the buffer must be restored to disk or not, and a 1 sector (currently 128 byte) character buffer.

DESCRIPTION:

```
struct buffer
```

```
{
```

```
    char    bfstore[b_p_sec];
```

size of disk buffer

```
    int     drive;
```

disk drive no.

```
    int     bfblk;
```

block number

```
    int     rsec;
```

relative sector in block

```
    int     bfflag;
```

dirty sector switch

```
};
```

NOTES:

Defined in disksystem.incl.

NAME:

cs_desc

PURPOSE:

This is a character set descriptor.

DESCRIPTION:

```
struct cs_desc
{
    int    ch_width;      Width of a character in dots
    int    ch_height;     Height of a character in dots (Must be not
                          greater than 16)
    int    *chset_ptr;    Pointer to vectors used in printing the
                          characters
    char    *effector_table; The effect of each character on format control,
                          e.g. new line, backspace, character delete,
                          etc.
};
```

NOTES:

Defined in cinc_structs.incl

A pointer to a cs_desc is generally referred to as a character set id.

NAME:

cs_desc

PURPOSE:

This is a character set descriptor.

DESCRIPTION:

```
struct cs_desc
{
    int    ch_width;           Width of a character in dots
    int    ch_height;          Height of a character in dots (Must be not
                                greater than 16)
    int    *chset_ptr;          Pointer to vectors used in printing the
                                characters
    char    *effector_table;    The effect of each character on format control,
                                e.g. new line, backspace, character delete,
                                etc.
    char    ch_cs_num;          If this character set is loaded into the
                                remote display head, this is the "variable"
                                number of the vector table.
    char    ch_eff_num;         If non-zero, this is the "variable" number of
                                this effector table in the remote display
                                heads. If zero, this character set is not
                                loaded into the display heads.
};
```

NOTES:

Defined in cinc_structs.incl

A pointer to a cs_desc is generally referred to as a character set id.

NAME: dev_entry

PURPOSE: this structure contains information about a device or file.

DESCRIPTION:

struct dev_entry	
{	
int owner;	The process ID of the owner of the device
struct queue_head *handler_q;	Pointer to the input queue of the handler process
struct semaphore dev_sem;	The controlling semaphore for the device.
struct semaphore *requestor;	Pointer to the semaphore which controls access to this device
struct req_block block;	A request block which may be used with this device. This block is used by the various io_sys routines when communicating a request to the device handler process.
int dindex;	Index block number if the device is actually a file.
int ddrive;	What drive a file resides on
};	

NOTES:

Defined in structures.incl

NAME:

dev_names

PURPOSE:

This is an array of pointers to character strings. It defines the "names" by which various resources of the io_system may be identified. It is used in conjunction with the integer array dev_types. The size of dev_names is controlled by the defined name fixed_names contained in the file disksystem.incl.

DESCRIPTION:

```
char    *dev_names[fixed_names]
{
    &"dev_tp",
    &"dev_kb",           Sample of permanent names used
    &"dev_ph",           in an IT operating system.
    &"dev_rx0",
    &"dev_rx1"
};
```

PARAMETERS:

fixed_names A define indicating the size of the table.

NOTES:

Defined in disk/disk_tables.c.

NAME:

dev_types

PURPOSE:

An integer array used in conjunction with dev_names which indicates the DEV_TAB slot to be used in referencing an IO device. If an entry in dev_types is negative, it represents the ones compliment of the DEV_TAB slot which is dedicated to the device. A positive entry indicates that corresponding name in dev_names is referencing a disk drive. The positive number is the number of the disk drive.

DESCRIPTION:

```
int    dev_types[fixed_names]
{
    -1, -2, -3, 0, 1
};
```

PARAMETERS:

The size of the table is determined by the define fixed_names in the file disksystem.incl.

NOTES:

Defined in disk/disk_tables.c.

NAME:

`dir_entry`

PURPOSE:

This defines the format of the information contained in a directory slot. An entry with a nonzero file type indicates that the particular file is a subdirectory and should be opened as such. The first `dir_entry` of a directory is a descriptor for the directory itself. In this case, the file type is the block number of the index block for the parent of the directory.

DESCRIPTION:

`struct dir_entry`

```
{
    int      filetype;           0 = plain file, else = directory
    int      filler[3];         add new stuff here
    int      mxblk;             max block in index
    int      mxoff;             offset in the highest block
    int      index;             index block for this file
    char      filename[b_p_name]; file name
    char      flgs;             flags, 1 = never delete
};
```

NOTES:

Defined in `disksystem.incl`.

NAME:

drive_info

PURPOSE:

This structure holds information about online drives. Contained therein is a copy of the drive freemap, the dir_entry of the root directory of the file at the time the disk was first brought on line, a file_info structure, and a buffer structure. The last two are used any time the root directory or the drive freemap must be modified.

DESCRIPTION:

struct drive_info

{

int freemap[free_sz];

freemap for the drive.

struct dir_entry dslot;

dir entry of the root

struct file_info dfib;

for accessing the directory

struct buffer denv;

environment to be used by disk for
directory or freemap reads/writes

};

NOTES:

Defined in disksystem.incl.

NAME:

`env_desc`

PURPOSE:

Used to store print environment descriptors.

DESCRIPTION:

```
struct env_desc
{
    struct cs_desc *e_cs;           This holds the character set ID
    struct page_desc e_page;       This holds the page descriptor
    int e_xcursor;                 The cursor x coordinate
    int e_ycursor;                 The cursor y coordinate
};
```

NOTES:

Defined in `cinc_structs.incl`

NAME:

env_desc

PURPOSE:

Used to store print environment descriptors.

DESCRIPTION:

struct env_desc

{

struct cs_desc *e_cs;

This holds the character set ID

struct page_desc e_page;

This holds the page descriptor

int e_xcursor;

The cursor x coordinate

int e_ycursor;

The cursor y coordinate

int e_pnl;

The set of currently selected remote
display heads

};

NOTES:

Defined in cinc_structs.incl

NAME:

file_info

PURPOSE:

File information block. This contains most of the information on a specific file. eg index block subset, disk pointers, file maxima and directory slot number.

DESCRIPTION:

```
struct file_info
{
    int      ndxs[ndxs_sz];      index subset
    int      ndxs_off;          offset in subset
    int      ndx_blk;           block number of index blk
    int      ndx_f;             index flag

    int      fblk;              file position block addr
    int      foff;              file position offset
    int      fblkno;            file position block number

    int      maxblk;            last block in index used
    int      maxoff;            max offset in last block
    int      dir_s_index;       index of directory for this file
    int      dir_s_off;         offset in directory
    int      fflags;            directory information flags
};
```

NOTES:

Defined in disksystem.incl.

NAME:

map

PURPOSE:

Defines a piece of free memory for use by the memory management routines

DESCRIPTION:

```
struct map
{
    char *m_size;      The size, in bytes, of a piece of free memory
    char *m_addr;      The address of the piece
};
```

NOTES:

Defined in structures.incl

NAME:

map

PURPOSE:

Defines a piece of free memory for use by the memory management routines

DESCRIPTION:

```
struct map
{
    int  m_size;           The size, in bytes, of a piece of free memory
    int *m_addr;          The address of the piece
};
```

NOTES:

Defined in structures.incl

NAME:

page_desc

PURPOSE:

An instance of this structure is used to describe the "page" (an area of the plasma panel) to be used for some function, typically printing.

DESCRIPTION:

```
struct page_desc
{
    int p_left;           Left edge of the page in dots from the left
                          edge of the screen
    int p_bottom;        Bottom edge of page in dots from the bottom
                          edge of the screen
    int p_width;         Width of the page in dots
    int p_height;        Height of the page in dots
};
```

NOTES:

Defined in cinc_structs.incl

NAME:

process

PURPOSE:

This structure contains information used by startup to create a process at system initialization time.

DESCRIPTION:

struct process

{

int (*procechure)();

A pointer to the entry point of the procedure to be started as a process

int size_of_stack;

The size, in words, of the stack that will be assigned to the process

int parm;

A one-word parameter to be passed to the procedure

int prty;

The process's priority

struct stack_base *id;

The process ID. This will be filled in by startup

};

NOTES:

Defined in structures.incl

NAME:

q_element

PURPOSE:

This is a single element in a queue.

DESCRIPTION:

struct q_element

{

int value;

struct q_element *link;

}

This is the one-word value which has
been placed in the queueA pointer to the next element in the
queue. The last element in the
queue points to the first element.

NOTES:

Defined in structures.incl

NAME:

queue_head

PURPOSE:

This is the header for a circular queue. Generally, any process may write to a queue, but only one process will read from it.

DESCRIPTION:

```
struct queue_head
```

```
{
```

```
    struct q_element *queue;
```

This is a pointer to the first element in the queue.

```
    struct semaphore sem;
```

This is the controlling semaphore for the queue. Read_q will pee this semaphore before reading from the queue.

Write_q will vee it after writing to the queue.

```
};
```

NOTES:

Defined in structures.incl

User programs should not modify fields in this structure.

NAME:

req_block

PURPOSE:

This is the format of a request to one of the device processes. Normally, request blocks are constructed by one of the io_sys routines (read, write, peek, etc.) and then the address of the block is sent to the device process.

DESCRIPTION:

```

struct req_block
{
    int type_req;          This is a code specifying the type of request.
                           The valid values and their mnemonics (defined
                           in constants.incl) are:
                           read_type      1
                           write_type    2
                           open_type     3
                           close_type    4
                           flush_type    5
                           peek_type     6
                           setmode_type  7
                           delete_type   9

    int data_len;          The actual length of data read or written by
                           this request.

    int buf_len;           The length of the user's buffer. By implication,
                           the length of the data he is asking to
                           transfer.

    char *buf_addr;        Pointer to the user's buffer

    struct semaphore *req_semaphore;
                           The requesting process does a pee on this
                           semaphore after sending the request. The
                           device process will vee it when the request
                           has been processed.

    int status_flags;      Device-dependent status information

    int id_field;          Equal to the DEV_TAB index minus the constant
                           first_disk. This is used to index disk tables.
};

```

NOTES:

Defined in structures.incl

NAME:

semaphore

PURPOSE:

This is the format of the IT implementation of Dijkstra's semaphores. They are used for synchronization between processes.

DESCRIPTION:

```
struct semaphore
{
```

```
    int count;
```

If this is greater than zero, it is the number of outstanding V's on the semaphore. If it is less than zero it is the number of processes blocked on the semaphore.

```
    struct q_element *ptr;
```

This is the start of a list of blocked processes. This linked list contains a pointer to the stack base of each process blocked on this semaphore.

```
};
```

NOTES:

Defined in structures.incl

Only the pee and vee kernel routines should alter these fields.

NAME:

stack_base

PURPOSE:

This is the information stored at the base of a process's stack.
It is used when switching processes.

DESCRIPTION:

```
struct stack_base
{
    int stack_size;      The size of the process's stack in words
    int R5;              Place to save register 5
    int R6;              Place to save register 6
    int stack_priority;  Priority of the process
    int guard_word;      This is a tag word (currently set to octal
                        0104401) which is used to check for stack
                        overflow.
};
```

NOTES:

Defined in structures.incl

NAME:

stack_base

PURPOSE:

This is the information stored at the base of a process's stack on the Level 6. It is used when switching processes.

DESCRIPTION:

```
struct stack_base
{
    int stack_size;      The size of the process's stack, in words
    int sv_B5;           The return address for the process
    int sv_B7;           Base register 7 - environment linkage
    int stack_priority;  The priority of this process.
    int guard_word;      This is a tag word (currently set to octal
                        0104401) which is used to check for stack
                        overflow.
};
```

NOTES:

Defined in structures.incl

NAME:

sv_regs

PURPOSE:

This is a template for the registers stored at entry to a Level 6 C program.

DESCRIPTION:

```
struct sv_regs
{
    int B7;      Saved value of base register 7
    int B6;      Saved value of base register 6
    int B5;      Saved value of base register 5
    int B3;      Saved value of base register 3
    int B2;      Saved value of base register 2
    int B1;      Saved value of base register 1
    int I;       Saved value of indicator register
    int R5;      Saved value of data register 5
    int R4;      Saved value of data register 4
    int R3;      Saved value of data register 3
    int R2;      Saved value of data register 2
    int R1;      Saved value of data register 1
    int M;       Saved value of M register
};
```

NOTES:

Defined in structures.incl

NAME:

target

PURPOSE:

This structure is used to define a touch target.

DESCRIPTION:

```
struct target
{
    int t_x;           X coordinate of lower left corner of target
                       in dots
    int t_y;           Y coordinate of lower left corner of target
                       in dots
    int twidth;        Width of target in dots
    int theight;       Height of target in dots
    int tvalue;        User supplied value for the target
    char *tlabel;      Pointer to a null-terminated character string
                       which is used to label the target.  If zero,
                       there is no label associated with this
                       target.
    int tcharset;      ID of the character set to be used when labeling
                       the target
    int tflags;        Miscellaneous flags.  These are the same values
                       that are input to the routine tt_create.
    int tother;        Reserved for future use
};
```

NOTES:

Defined in cinc_structs.incl

APPENDIX D

Character Set Description

This Appendix contains three tables which detail the character set for the plasma panel used by the IT. The first table lists all 128 of the ASCII characters and describes the graphic used in printing each one. The second table gives a detailed explanation of the format effector characters in the standard ASCII charset. The last table indicates the values which are used in the Effector Tables for IT charsets.

8/30/77

Table of Characters Displayed on the Plasma Panel

Octal	ASCII Name	Character Graphic	Octal	ASCII Name	Character Graphic	Octal	ASCII Name	Character Graphic
000	NUL	<Nothing>	053	+	+	126	V	V
001	SOH	Inverted A	054	,	,	127	W	W
002	STX	Inverted B	055	-	-	130	X	X
003	ETX	Inverted C	056	.	.	131	Y	Y
004	EOT	Inverted D	057	/	/	132	Z	Z
005	ENQ	Inverted E	060	0	0	133	[[
006	ACK	Inverted F	061	1	1	134	\	\
007	BEL	Inverted G *	062	2	2	135]]
010	BS	Inverted H *	063	3	3	136	^	^
011	HT	Inverted I *	064	4	4	137	~	~
012	LF	Inverted J *	064	5	5	140		
013	VT	Inverted K *	066	6	6	141	a	a
014	FF	Inverted L *	067	7	7	142	b	b
015	CR	Inverted M *	070	8	8	143	c	c
016	RRS	Inverted N	071	9	9	144	d	d
017	BRS	Inverted O	072	:	:	145	e	e
020	DLE	Inverted P	073	;	;	146	f	f
021	DC1	Inverted Q	074	<	<	147	g	g
022	DC2	Inverted R	075	=	=	150	h	h
023	DC3	Inverted S	076	>	>	151	i	i
024	DC4	Inverted T	077	?	?	152	j	j
025	NAK	Inverted U	100	@	@	153	k	k
026	SYN	Inverted V	101	A	A	154	l	l
027	ETB	Inverted W	102	B	B	155	m	m
030	CAN	Inverted X *	103	C	C	156	n	n
031	EM	Inverted Y	104	D	D	157	o	o
032	SUB	Inverted Z	105	E	E	160	p	p
033	ESC	Inverted 3	106	F	F	161	q	q
034	FS	Inverted 7	107	G	G	162	r	r
035	GS	Inverted 0	110	H	H	163	s	s
036	RS	Inverted 8	111	I	I	164	t	t
037	US	Inverted n	112	J	J	165	u	u
040	Space	Blank	113	K	K	166	v	v
041	!	!	114	L	L	167	w	w
042	"	"	115	M	M	170	x	x
043	#	#	116	N	N	171	y	y
044	\$	\$	117	O	O	172	z	z
045	%	%	120	P	P	173	{	{
046	&	&	121	Q	Q	174		
047	'	'	122	R	R	175	}	}
050	((123	S	S	176	~	~
051))	124	T	T	177	DEL	Box *
052	*	*	125	U	U			

"Inverted" characters are printed dark-on-light instead of the normal light-on-dark.

*These characters are format effectors in the standard character set. For their effect, see the next table.

8/30/77

IT Plasma Panel Format Effectors

Octal	ASCII Name	Description of Effect*
000	NUL	Does nothing.
007	BEL	Rings the bell.
010	BS	Moves the cursor one character to the left.
011	TAB	Moves the cursor to the right to the next 8-character tab stop.
012	LF	Moves the cursor to the start of the next line.
013	VT	Moves the cursor up one line.
014	FF	Clears the page and moves the cursor to the top left corner.
015	CR	Moves the cursor to the start of the current line.
030	CAN	Moves the cursor back to the start of the current line and erases the line.
177	DEL	Moves the cursor one character to the left and erases that character.

*All format effectors work in the subset of the plasma panel which has been specified as the current page.

Character Set Effector Code Values

Value**	Effect
0	Treat the character as an ordinary graphic.
1	Move the cursor to the left one character. (Backspace)
2	Ring the bell.
3	Move the cursor to the start of the current line. (Carriage return)
4	Move the cursor one character to the left and delete that character.
5	Clear the page and move the cursor to the top left.
6	Erase the current line.
7	Move the cursor down one line. (Line feed)
8	Move the cursor to the start of the next line. (New line)
9	Move the cursor to the right to the next tab stop. (Horizontal tab)
10	Move the cursor down to the next vertical tab stop. (Vertical tab)
11	Ignore the character. Do not print it or effect the format.

**These are the values stored in the character set effector table. If the meaning of these values must be changed, then the effector table and the routine put_ascii must both be changed.

APPENDIX E

Interfaces: Plasma and Touch Panels

PLASMA PANEL INTERFACE

This section describes the interface to the plasma display panel and gives an example of its use.

The plasma panel is controlled from the LSI-11 using four registers. The Control and Status Register, DISPCSR, is for general control. It selects the operation (write, erase) and mode for the interface. The X and Y address registers, DISPXADR and DISPYADR, specify the address of the point or group of points to be operated on. The Display Data Register, DISPDATA, is used to select a subset of a 16-dot group to operate on in parallel mode.

Modes of Operation

There are two modes of operation of the plasma panel interface: serial and parallel.

Serial mode - The serial mode of operation is used to operate on a single point on the display. In this mode, the point is addressed by the X and Y address registers. The operation to be performed is specified in the Control and Status Register. The contents of the Display Data Register are meaningless in the serial mode.

Parallel mode - The parallel mode of operation is used to operate simultaneously on 16 points of the display. In this mode, the X and Y address registers are used to address a group of 16 points. The contents of the Display Data Register act as a mask to select which of the 16 points will be effected by the current operation.

The contents of the X and Y address registers address the lower end of a vertical group of 16 points on the display screen. Each point in the group of 16 which corresponds to a

one in the Display Data Register will be affected by the current operation. Those bits which correspond to zeroes in the Display Data Register will not be affected. The low order bit of the Display Data Register corresponds to the lowest of the 16 vertical points being addressed. Note that the low-order 4 bits of the Y-address are treated as zeroes. Thus, every operation in the parallel mode addresses a group of 16 points whose lower end is on a 0 module 16 boundary. The full X-address is used.

Addressing Modes

There are four different addressing modes used to access the display panel interface. The addressing mode determines exactly when the operation specified in the Control and Status Register will be done.

Immediate mode. In immediate mode, the operation is performed immediately upon loading of the CSR. (In the current interface, this mode does not work properly.)

After X mode. In this mode, the specified operation is performed each time the X-address register is loaded.

After Y mode. The operation is performed each time the Y-address register is loaded.

After data mode. The operation is performed each time the Display Data Register is loaded. This is allowed in serial mode; however the contents of the data register will be ignored.

Registers

The addresses of the panel registers, and the meanings of the various bits are diagrammed below. The register addresses can be changed by altering the settings of the address selection switches. However, any programs written that refer to those addresses must be modified accordingly if these switch settings are changed.

DISPCSR - 177000

ERR	—	—	—	—	—	—	—	DONE	IE	PRT	BULK	PAR	F2	F1	F0
-----	---	---	---	---	---	---	---	------	----	-----	------	-----	----	----	----

DISPXADR - 177002

—	—	—	—	—	—	—	X8	X7	X6	X5	X4	X3	X2	X1	X0
---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----

DISPYADR - 177004

—	—	—	—	—	—	—	Y8	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----

DISPDATA - 177006

PD15	←														→	PD0
------	---	--	--	--	--	--	--	--	--	--	--	--	--	--	---	-----

Control and Status Register (DISPCSR) 177000

The control and status register is used to provide user-defined command and status functions for the display unit.

DISPCSR Bit Assignments

<u>BIT</u>	<u>NAME</u>	<u>FUNCTION</u>
15	ERROR	Not implemented in the current interface design but is provided for future versions. (Read Only)
14-08		Not used
07	DONE	Indicates that the display unit has executed the previously defined operation and is ready to continue.
06	INTENB	Interrupt Enable...not implemented in the current interface.
05	PRINT	Not implemented in the current interface.

<u>BIT</u>	<u>NAME</u>	<u>FUNCTION</u>
04	BULK	Bulk erase command to the display (screen clear). (Read/Write)
03	PARALLEL	When set, places the interface and display in the parallel mode of operation. (Read/Write)
02-00	F2-F0	Function bits. Used to define the operation to be performed on the display. Table #1 describes these operations. (Read/Write)

TABLE 1 - Display Operations

<u>FUNCTION BITS</u>			<u>OPERATION</u>
F2	F1	F0	
0	0	0	ERASE "IMMEDIATE"
0	0	1	ERASE AFTER LOADING X-ADDR REGISTER
0	1	0	ERASE AFTER LOADING Y-ADDR REGISTER
0	1	1	ERASE AFTER LOADING DATA REGISTER
1	0	0	WRITE "IMMEDIATE"
1	0	1	WRITE AFTER LOADING X-ADDR REGISTER
1	1	0	WRITE AFTER LOADING Y-ADDR REGISTER
1	1	1	WRITE AFTER LOADING DATA REGISTER

Parallel Data Register (DISPDATA) 177006

The parallel data register is a 16-bit read/write register that may be read or loaded from the processor bus. Information from the bus is loaded into this register under program control.

A "one" in any bit position (in conjunction with X-ADDR and Y-ADDR) indicates that the addressed point on the display is to be either illuminated (written) or extinguished (erased) depending on the contents

of the DISPCSR. The contents of this register have no effect when the interface is operating in the "SERIAL" mode.

X-ADDRESS Register (DISPXADR) 177002

The X-Address register is a 16-bit read/write register that may be read or loaded from the processor bus. Information from the bus is loaded into this register under program control.

This register is used to specify the 9-bit X-Address to be operated upon on the display. The address should be right-justified in the register.

Y-Address Register (DISPYADR) 177004

The Y-Address register is a 16-bit read/write register that may be read or loaded from the processor bus. Information from the bus is loaded into this register under program control.

This register is used to specify the 9-bit Y-Address to be operated upon on the display. This address should be right-justified on the register.

Example of Use

The following section of code will draw the outline of a 16x16 dot box on the plasma display panel. The lower, left-hand corner of the box will be at address (X,Y). The example is intended to be more illustrative than efficient.


```
#define DISPCSR (0177000 -> word)
#define DISPXADR (0177002 -> word)
#define DISPYADR (0177004 -> word)
#define DISPDATA (0177006 -> word)
```

```
struct {int word};
```

```
box()
```

```
{
```

```
int xad, yad;
```

```
while ((DISPCSR & 0200) == 0); /* loop until the done bit is on to be
                                sure that the previous operation is
                                completed */
```

```
/*
 * Use the serial, write-after-y modes to draw the
 * sides of the box
 */
```

```
DISPCSR = 3; /* set to serial, write after y mode */
DISPXADR = X;
```

```
for (yad = Y; yad < Y + 16; ++yad)
{
    DISPYADR = yad; /* draw the left side of the box */
                    /* this will cause one dot to be
                    written */
    while ((DISPCSR & 0200) == 0); /* wait for operation to complete */
}
```

```
DISPXADR = X + 15; /* set x address to write right side of
                    box */
```

```
for (yad = Y; yad < Y + 16; ++yad)
{
    DISPYADR = yad; /* draw the right side of the box */
                    /* write one dot */
    while ((DISPCSR & 0200) == 0); /* wait for completion */
}
```

```
/*
 * Use the parellel, write after x modes to write the top and
 * bottom of the box at the same time
 */
```

```
DISPCSR = 015; /* Put the panel into parallel, write
                after x mode */
DISPDATA = 0100001; /* This will cause each write to effect
                    the top and bottom dots in each
                    16-dot group */
```

```
DISPYADR = Y;
```

```
for (xad = X + 1; xad < X + 16; ++xad)
{
    DISPXADR = xad; /* write one dot at top and one at
                    bottom of the box */
    while ((DISPCSR & 0200) == 0); /* wait for completion */
}
```

```
}
```


TOUCH PANEL INTERFACE

The touch panel is interfaced to the LSI-11 through a standard DEC DRV11 interface. When a touch is made, the panel will send two characters to the processor via the data register of the interface. The touch coordinates are encoded in the data register as:

—	—	—	X4	X3	X2	X1	X0	—	—	—	Y4	Y3	Y2	Y1	Y0
---	---	---	----	----	----	----	----	---	---	---	----	----	----	----	----

An X coordinate of zero corresponds to the left of the screen, and a Y coordinate of zero corresponds to the bottom of the screen.

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS
BEFORE COMPLETING FORM

1. REPORT NUMBER CCTC-WAD Document #7616 CAC Document #236		2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Research in Network Data Management and Resource Sharing INTELLIGENT TERMINAL PROGRAMMER'S MANUAL (Vol. 2)		5. TYPE OF REPORT & PERIOD COVERED Research	
		6. PERFORMING ORG. REPORT NUMBER CAC Document #236	
7. AUTHOR(s) Deborah S. Brown, Daniel J. Koptezky, John R. Mullen, and David A. Willcox		8. CONTRACT OR GRANT NUMBER(s) DCA100-76-C-0088	
9. PERFORMING ORGANIZATION NAME AND ADDRESS Center For Advanced Computation, University of Illinois at Urbana-Champaign Urbana, Illinois 61801		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS	
11. CONTROLLING OFFICE NAME AND ADDRESS Command and Control Technical Center Defense Communications Agency 11440 Isaac Newton Sq., Reston VA 22090		12. REPORT DATE October 31, 1977	
		13. NUMBER OF PAGES 169	
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED	
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE	
16. DISTRIBUTION STATEMENT (of this Report) No Restriction on Distribution			
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)			
18. SUPPLEMENTARY NOTES Copies of this report may be obtainde from (11), above.			
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Intelligent Terminals			
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) Volume 2 of the Programmer's Manual for the touch-oriented Intelligent Terminal developed for CCTC-WAD's Man-Machine Interface Project.			

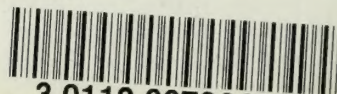
UNIVERSITY OF ILLINOIS-URBANA

510.841L63C

C001

CAC DOCUMENT\$URBANA

235-236 1977



3 0112 007264069